# *Introduction to*

# JAVA™

## PROGRAMMING AND DATA STRUCTURES

COMPREHENSIVE VERSION

Y. DANIEL LIANG

Pearson

12th Edition

CHAPTER

9

# OBJECTS AND CLASSES

## Objectives

- To describe objects and classes, and use classes to model objects (§9.2).
- To use UML graphical notation to describe classes and objects (§9.2).
- To demonstrate how to define classes and create objects (§9.3).
- To create objects using constructors (§9.4).
- To define a reference variable using a reference type and access objects via object reference variables (§9.5).
- To access an object's data and methods using the object member access operator (`.`) (§9.5.1).
- To define data fields of reference types and assign default values for an object's data fields (§9.5.2).
- To distinguish between object reference variables and primitive-data-type variables (§9.5.3).
- To use the Java library classes `Date`, `Random`, and `Point2D` (§9.6).
- To distinguish between instance and static variables and methods (§9.7).
- To define private data fields with appropriate getter and setter methods (§9.8).
- To encapsulate data fields to make classes easy to maintain (§9.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§9.10).
- To store and process objects in arrays (§9.11).
- To create immutable objects from immutable classes to protect the contents of objects (§9.12).
- To determine the scope of variables in the context of a class (§9.13).
- To use the keyword `this` to refer to the calling object itself (§9.14).

## 9.1 Introduction


*Object-oriented programming enables you to develop large-scale software and GUIs effectively.*

Object-oriented programming is essentially a technology for developing reusable software. Having learned the material in the preceding chapters, you are able to solve many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems. Suppose you want to develop a graphical user interface (GUI, pronounced *goo-ee*) as shown in Figure 9.1. How would you program it?

why OOP?

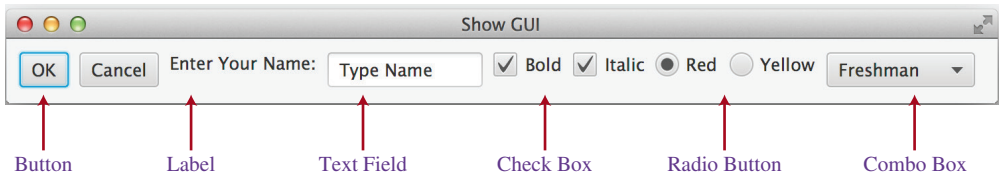



**FIGURE 9.1** The GUI objects are created from classes.

This chapter introduces object-oriented programming, which you can use to develop GUI and large-scale software systems.

## 9.2 Defining Classes for Objects


*A class defines the properties and behaviors for objects.*

VideoNote
Define classes and create objects

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, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

object
state of an object
properties
attributes
data fields
behavior
actions

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field `radius`, which is the property that characterizes a circle. A rectangle object, for example, has the data fields `width` and `height`, which are the properties that characterize a rectangle.
- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, you may define methods named `getArea()` and `getPerimeter()` for circle objects. A circle object may invoke `getArea()` to return its area and `getPerimeter()` to return its perimeter. You may also define the `setRadius(radius)` method. A circle object can invoke this method to change its radius.

class
contract

instantiation
instance

Objects of the same type are defined using a common class. A *class* is a template, blueprint, or *contract* that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. 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 that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe. Figure 9.2 shows a class named `Circle` and its three objects.

data field
method
constructors

A Java class uses variables to define data fields and methods to define actions. In addition, a class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Figure 9.3 shows an example of defining the class for circle objects.

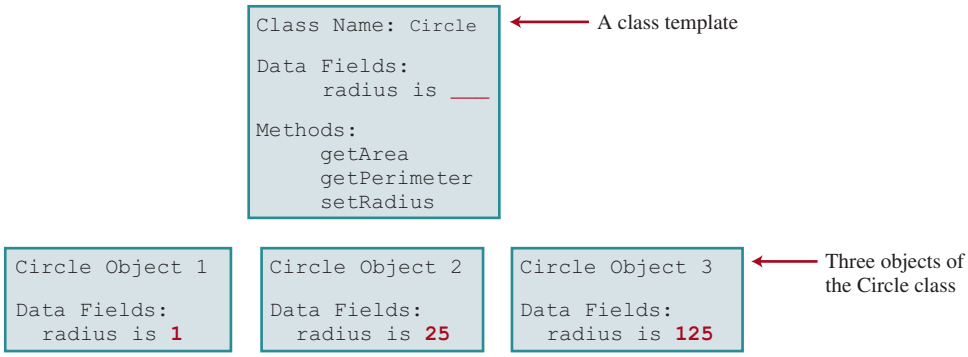



**FIGURE 9.2** A class is a template for creating objects.

```
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

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

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Data fields

Constructors

Methods

**FIGURE 9.3** A class is a construct that defines objects of the same type.

The **`Circle`** class is different from all of the other classes you have seen thus far. It does not have a **`main`** method, and therefore cannot be run; it is merely a definition for circle objects. The class that contains the **`main`** method will be referred to in this book, for convenience, as the *main class*.

main class

The illustration of class templates and objects in Figure 9.2 can be standardized using *Unified Modeling Language (UML)* notation. This notation, as shown in Figure 9.4, is called a *UML class diagram*, or simply a *class diagram*. In the class diagram, the data field is denoted as

Unified Modeling Language (UML)

class diagram

```
dataFieldName: dataFieldType
```

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

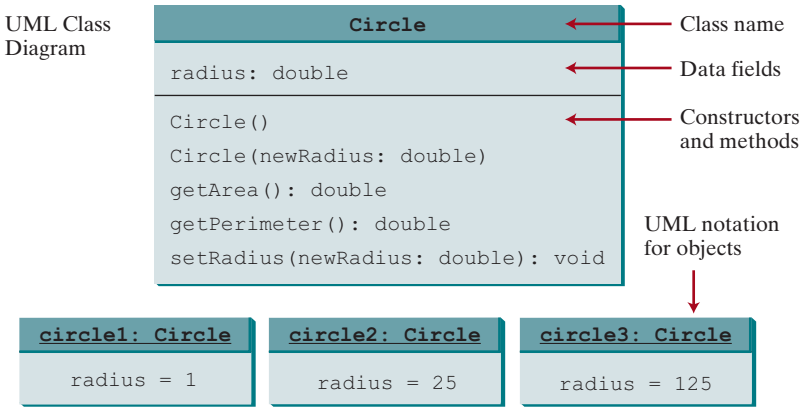



**FIGURE 9.4** Classes and objects can be represented using UML notation.

The method is denoted as

```
methodName(parameterName: parameterType): returnType
```

## 9.3 Example: Defining Classes and Creating Objects


*Classes are definitions for objects and objects are created from classes.*

This section gives two examples of defining classes and uses the classes to create objects. Listing 9.1 is a program that defines the `Circle` class and uses it to create objects. The program constructs three circle objects with radius `1`, `25`, and `125` and displays the `radius` and `area` of each of the three circles. It then changes the radius of the second object to `100` and displays its new radius and area.

### LISTING 9.1 TestCircle.java

main class

main method

create object

create object

create object

```
1  public class TestCircle {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a circle with radius 1
5      Circle circle1 = new Circle();
6      System.out.println("The area of the circle of radius "
7        + circle1.radius + " is " + circle1.getArea());
8
9      // Create a circle with radius 25
10     Circle circle2 = new Circle(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + "is" + circle2.getArea());
13
14     // Create a circle with radius 125
15     Circle circle3 = new Circle(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100; // or circle2.setRadius(100)
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea());
23   }
24 }
25
```

```
26  // Define the circle class with two constructors
27  class Circle {                                              class Circle
28    double radius;                                            data field
29
30    /** Construct a circle with radius 1 */
31    Circle() {                                                no-arg constructor
32      radius = 1;
33    }
34
35    /** Construct a circle with a specified radius */
36    Circle(double newRadius) {                                second constructor
37      radius = newRadius;
38    }
39
40    /** Return the area of this circle */
41    double getArea() {                                        getArea
42      return radius * radius * Math.PI;
43    }
44
45    /** Return the perimeter of this circle */
46    double getPerimeter() {                                   getPerimeter
47      return 2 * radius * Math.PI;
48    }
49
50    /** Set a new radius for this circle */
51    void setRadius(double newRadius) {                        setRadius
52      radius = newRadius;
53    }
54  }
```

```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

The program contains two classes. The first of these, `TestCircle`, is the main class. Its sole purpose is to test the second class, `Circle`. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the `main` method in the main class. client

You can put the two classes into one file, but only one class in the file can be a *public class*. Furthermore, the public class must have the same name as the file name. Therefore, the file name is **TestCircle.java**, since `TestCircle` is public. Each class in the source code is compiled into a **.class** file. When you compile **TestCircle.java**, two class files **TestCircle.class** and **Circle.class** are generated, as shown in Figure 9.5. public class

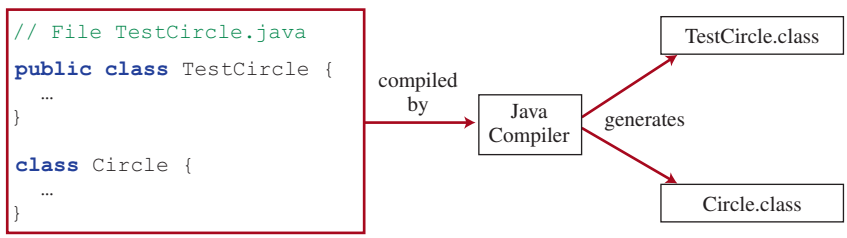



**FIGURE 9.5** Each class in the source code file is compiled into a **.class** file.

The main class contains the `main` method (line 3) that creates three objects. As in creating an array, the `new` operator is used to create an object from the constructor: `new Circle()` creates an object with radius `1` (line 5), `new Circle(25)` creates an object with radius `25` (line 10), and `new Circle(125)` creates an object with radius `125` (line 15).

These three objects (referenced by `circle1`, `circle2`, and `circle3`) have different data but the same methods. Therefore, you can compute their respective areas by using the `getArea()` method. The data fields can be accessed via the reference of the object using `circle1.radius`, `circle2.radius`, and `circle3.radius`, respectively. The object can invoke its method via the reference of the object using `circle1.getArea()`, `circle2.getArea()`, and `circle3.getArea()`, respectively.

These three objects are independent. The radius of `circle2` is changed to `100` in line 20. The object's new radius and area are displayed in lines 21 and 22.

There are many ways to write Java programs. For instance, you can combine the two classes in the preceding example into one, as given in Listing 9.2.

## LISTING 9.2 Circle.java (AlternativeCircle.java)

```
1  public class Circle {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a circle with radius 1
5      Circle circle1 = new Circle();
6      System.out.println("The area of the circle of radius "
7        + circle1.radius + " is " + circle1.getArea());
8
9      // Create a circle with radius 25
10     Circle circle2 = new Circle(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + " is " + circle2.getArea());
13
14     // Create a circle with radius 125
15     Circle circle3 = new Circle(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100;
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea());
23   }
24
25   double radius;
26
27   /** Construct a circle with radius 1 */
28   Circle() {
29     radius = 1;
30   }
31
32   /** Construct a circle with a specified radius */
33   Circle(double newRadius) {
34     radius = newRadius;
35   }
36
37   /** Return the area of this circle */
38   double getArea() {
39     return radius * radius * Math.PI;
40   }
41
```

main method

data field

no-arg constructor

second constructor

method

```
42    /** Return the perimeter of this circle */
43    double getPerimeter() {
44      return 2 * radius * Math.PI;
45    }
46
47    /** Set a new radius for this circle */
48    void setRadius(double newRadius) {
49      radius = newRadius;
50    }
51  }
```

Since the combined class has a `main` method, it can be executed by the Java interpreter. The `main` method is the same as that in Listing 9.1. This demonstrates that you can test a class by simply adding a `main` method in the same class.

As another example, consider television sets. Each TV is an object with states (current channel, current volume level, and power on or off) and behaviors (change channels, adjust volume, and turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 9.6.

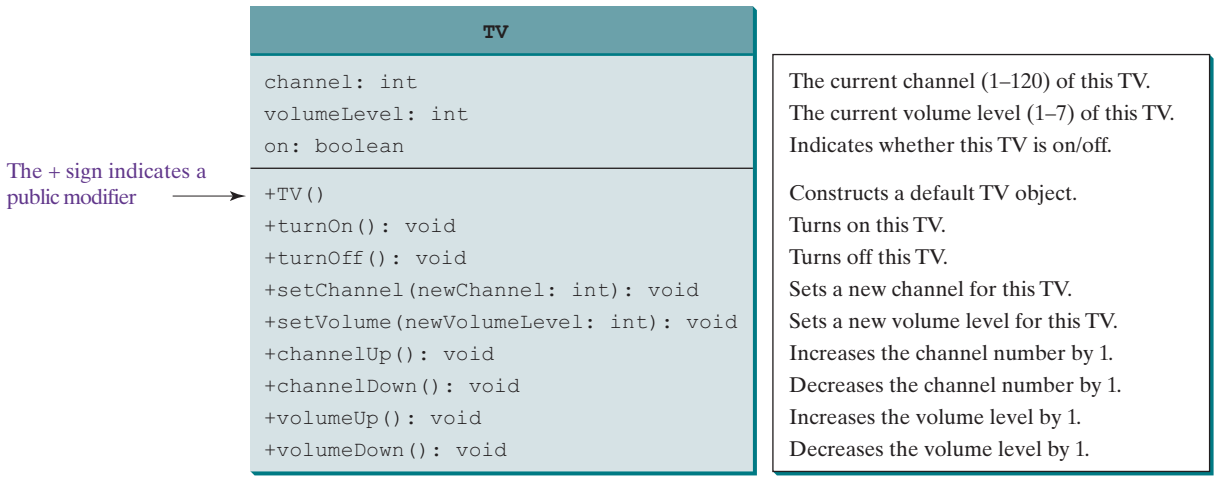



**FIGURE 9.6** The TV class models TV sets.

Listing 9.3 gives a program that defines the `TV` class.

## LISTING 9.3 TV.java

```
1   public class TV {
2     int channel = 1; // Default channel is 1                    data fields
3     int volumeLevel = 1; // Default volume level is 1
4     boolean on = false; // TV is off
5
6     public TV() {                                               constructor
7     }
8
9     public void turnOn() {                                      turn on TV
10      on = true;
11    }
12
13    public void turnOff() {                                     turn off TV
```

```
14      on = false;
15    }
16
17    public void setChannel(int newChannel) {
18      if (on && newChannel >= 1 && newChannel <= 120)
19        channel = newChannel;
20    }
21
22    public void setVolume(int newVolumeLevel) {
23      if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
24        volumeLevel = newVolumeLevel;
25    }
26
27    public void channelUp() {
28      if (on && channel < 120)
29        channel++;
30    }
31
32    public void channelDown() {
33      if (on && channel > 1)
34        channel--;
35    }
36
37    public void volumeUp() {
38      if (on && volumeLevel < 7)
39        volumeLevel++;
40    }
41
42    public void volumeDown() {
43      if (on && volumeLevel > 1)
44        volumeLevel--;
45    }
46  }
```

set a new channel

set a new volume

increase channel

decrease channel

increase volume

decrease volume

The constructor and methods in the **TV** class are defined public so they can be accessed from other classes. Note the channel and volume level are not changed if the TV is not on. Before either of these is changed, its current value is checked to ensure it is within the correct range.

Listing 9.4 gives a program that uses the **TV** class to create two objects.

## LISTING 9.4 TestTV.java

```
1  public class TestTV {
2    public static void main(String[] args) {
3      TV tv1 = new TV();
4      tv1.turnOn();
5      tv1.setChannel(30);
6      tv1.setVolume(3);
7
8      TV tv2 = new TV();
9      tv2.turnOn();
10     tv2.channelUp();
11     tv2.channelUp();
12     tv2.volumeUp();
13
14     System.out.println("tv1's channel is " + tv1.channel
15       + " and volume level is " + tv1.volumeLevel);
16     System.out.println("tv2's channel is " + tv2.channel
17       + " and volume level is " + tv2.volumeLevel);
18   }
19 }
```

main method

create a TV

turn on

set a new channel

set a new volume

create a TV

turn on

increase channel

increase volume

display state

```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

The program creates two objects in lines 3 and 8 and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 14–17. The methods are invoked using syntax such as `tv1.turnOn()` (line 4). The data fields are accessed using syntax such as `tv1.channel` (line 14).

These examples have given you a glimpse of classes and objects. You may have many questions regarding constructors, objects, reference variables, accessing data fields, and invoking object's methods. The sections that will follow discuss these issues in detail.

**9.3.1** Describe the relationship between an object and its defining class.

**9.3.2** How do you define a class?

**9.3.3** How do you declare an object's reference variable?

**9.3.4** How do you create an object?

## 9.4 Constructing Objects Using Constructors

*A constructor is invoked to create an object using the* `new` *operator.*

Constructors are a special kind of method. They have three peculiarities:

- A constructor must have the same name as the class itself. — constructor's name
- Constructors do not have a return type—not even `void`. — no return type
- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects. — `new` operator

The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values. — overloaded constructors

It is a common mistake to put the `void` keyword in front of a constructor. For example, — no `void`

```
public void Circle() {
}
```

In this case, `Circle()` is a method, not a constructor.

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows: — constructing objects

```
new ClassName(arguments);
```

For example, `new Circle()` creates an object of the `Circle` class using the first constructor defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor defined in the `Circle` class.

A class normally provides a constructor without arguments (e.g., `Circle()`). Such a constructor is referred to as a *no-arg* or *no-argument constructor*. — no-arg constructor

A class may be defined without constructors. In this case, a public no-arg constructor with an empty body is implicitly defined in the class. This constructor, called a *default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*. — default constructor

**9.4.1** What are the differences between constructors and methods?

**9.4.2** When will a class have a default constructor?

**9.4.3** What is wrong with each of the following programs?

```
1  public class ShowErrors {
2    public static void main(String[] args) {
3      ShowErrors t = new ShowErrors(5);
4    }
5  }
```

(a)

```
1  public class ShowErrors {
2    public static void main(String[] args) {
3      C c = new C(5.0);
4      System.out.println(c.value);
5    }
6  }
7
8  class C {
9    int value = 2;
10 }
```

(b)

**9.4.4** What is wrong in the following code?

```
1  class Test {
2    public static void main(String[] args) {
3      A a = new A();
4      a.print();
5    }
6  }
7
8  class A {
9    String s;
10
11   A(String newS) {
12     s = newS;
13   }
14
15   public void print() {
16     System.out.print(s);
17   }
18 }
```

## 9.5 Accessing Objects via Reference Variables


*An object's data and methods can be accessed through the dot ( . ) operator via the object's reference variable.*

Newly created objects are allocated in the memory. They can be accessed via reference variables.

reference variable

Objects are accessed via the object's *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

reference type

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

```
Circle myCircle;
```

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

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

You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

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

The variable `myCircle` holds a reference to a `Circle` object.

**Note**

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 most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that `myCircle` is a `Circle` object rather than use the long-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.

object vs. object reference variable

**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.

array object

## 9.5.1 Accessing an Object's Data and Methods

In OOP terminology, an object's member refers to its data fields and methods. After an object is created, its data can be accessed and its methods can be invoked using the *dot operator* (`.`), also known as the *object member access operator*:

dot operator ( . )

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.

For example, `myCircle.radius` references the radius in `myCircle` and `myCircle.getArea()` invokes the `getArea` method on `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 `getArea` is referred to as an *instance method* because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

instance variable

instance method

calling object

**Caution**

Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`) to invoke a method in the `Math` class. Can you invoke `getArea()` using `Circle.getArea()`? The answer is no. All the methods in the `Math` class are static methods, which are defined using the `static` keyword. However, `getArea()` is an instance method, and thus nonstatic. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`). Further explanation will be given in Section 9.7, Static Variables, Constants, and Methods.

invoking methods

**Note**

Usually you create an object and assign it to a variable, then later you can use the variable to reference the object. 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 using the syntax:

```
new Circle();
```

or

```
System.out.println("Area is " + new Circle(5).getArea());
```

The former statement creates a `Circle` object. The latter creates a `Circle` object and invokes its `getArea` method to return its area. An object created in this way is known as an *anonymous object*.

anonymous object

## 9.5.2 Reference Data Fields and the null Value

reference data fields

The data fields can be of reference types. For example, the following `Student` class contains a data field `name` of the `String` type. `String` is a predefined Java class.

```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

null value

If a data field of a reference type does not reference any object, the data field holds a special Java value, `null`. `null` is a literal just like `true` and `false`. While `true` and `false` are Boolean literals, `null` is a literal for a reference type. `null` is not a Java keyword, but it is a reserved word in Java.

default field values

The default value of a data field is `null` for a reference type, `0` for a numeric type, `false` for a `boolean` type, and `\u0000` for a `char` type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of the data fields `name`, `age`, `isScienceMajor`, and `gender` for a `Student` object:

```
class TestStudent {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " + student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

The following code has a compile error, because the local variables `x` and `y` are not initialized:

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

NullPointerException

**Caution**

`NullPointerException` is a common runtime error. It occurs when you invoke a method on a reference variable with a `null` value. Make sure you assign an object reference to the variable before invoking the method through the reference variable (see CheckPoint Question 9.5.5c).

## 9.5.3 Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. 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 9.7, the value of `int` variable `i` is `int` value `1`, and the value of `Circle` object `c` holds a reference to where the contents of the `Circle` object are stored in memory.

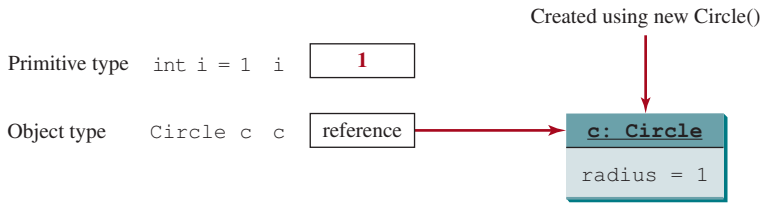



**FIGURE 9.7** A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

When you assign one variable to 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 9.8, the assignment statement `i = j` copies the contents of `j` into `i` for

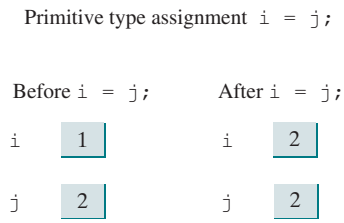



**FIGURE 9.8** Primitive variable `j` is copied to variable `i`.

primitive variables. As shown in Figure 9.9, the assignment statement `c1 = c2` copies the reference of `c2` into `c1` for reference variables. After the assignment, variables `c1` and `c2` refer to the same object.

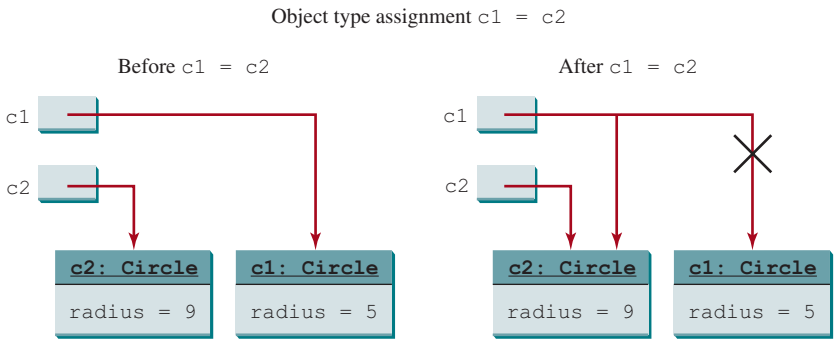



**FIGURE 9.9** Reference variable `c2` is copied to variable `c1`.

### Note

As illustrated in Figure 9.9, 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 and therefore is now known as *garbage*. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

garbage

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 JVM will automatically collect the space if the object is not referenced by any reference variable.


**9.5.1** Is an array an object or a primitive-type value? Can an array contain elements of an object type? Describe the default value for the elements of an array.

**9.5.2** Which operator is used to access a data field or invoke a method from an object?

**9.5.3** What is an anonymous object?

**9.5.4** What is `NullPointerException`?

**9.5.5** What is wrong with each of the following programs?

```
1  public class ShowErrors {
2    public static void main(String[] args) {
3      ShowErrors t = new ShowErrors();
4      t.x();
5    }
6  }
```

(a)

```
1  public class ShowErrors {
2    public void method1() {
3      Circle c;
4      System.out.println("What is radius "
5        + c.getRadius());
6      c = new Circle();
7    }
8  }
```

(b)

**9.5.6** What is the output of the following code?

```
public class A {
  boolean x;

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

## 9.6 Using Classes from the Java Library


*The Java API contains a rich set of classes for developing Java programs.*

Listing 9.1 defined the `Circle` class and created objects from the class. You will frequently use the classes in the Java library to develop programs. This section gives some examples of the classes in the Java library.

### 9.6.1 The `Date` Class

In Listing 2.7, ShowCurrentTime.java, you learned how to obtain the current time using `System.currentTimeMillis()`. You used the division and remainder operators to extract the current second, minute, and hour. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class, as shown in Figure 9.10.

`java.util.Date` class

| java.util.Date | |
|---|---|
| +Date() | Constructs a Date object for the current time. |
| +Date(elapseTime: long) | Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT. |
| +toString(): String | Returns a string representing the date and time. |
| +getTime(): long | Returns the number of milliseconds since January 1, 1970, GMT. |
| +setTime(elapseTime: long): void | Sets a new elapse time in the object. |

**FIGURE 9.10** A `Date` object represents a specific date and time.

You can use the no-arg constructor in the **Date** class to create an instance for the current date and time, the **getTime()** method to return the elapsed time in milliseconds since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string, For example, the following code:

```
java.util.Date date = new java.util.Date();                    create object
System.out.println("The elapsed time since Jan 1, 1970 is " +
  date.getTime() + " milliseconds");                           get elapsed time
System.out.println(date.toString());                           invoke toString
```

displays the output as follows:

```
The elapsed time since Jan 1, 1970 is 1324903419651 milliseconds
Mon Dec 26 07:43:39 EST 2011
```

The **Date** class has another constructor, **Date(long elapseTime)**, which can be used to construct a **Date** object for a given time in milliseconds elapsed since January 1, 1970, GMT.

## 9.6.2 The Random Class

You have used **Math.random()** to obtain a random **double** value between **0.0** and **1.0** (excluding **1.0**). Another way to generate random numbers is to use the **java.util.Random** class, as shown in Figure 9.11, which can generate a random **int**, **long**, **double**, **float**, and **boolean** value.

| java.util.Random | |
|---|---|
| +Random() | Constructs a Random object with the current time as its seed. |
| +Random(seed: long) | Constructs a Random object with a specified seed. |
| +nextInt(): int | Returns a random int value. |
| +nextInt(n: int): int | Returns a random int value between 0 and n (excluding n). |
| +nextLong(): long | Returns a random long value. |
| +nextDouble(): double | Returns a random double value between 0.0 and 1.0 (excluding 1.0). |
| +nextFloat(): float | Returns a random float value between 0.0F and 1.0F (excluding 1.0F). |
| +nextBoolean(): boolean | Returns a random boolean value. |

**FIGURE 9.11** A **Random** object can be used to generate random values.

When you create a **Random** object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The no-arg constructor creates a **Random** object using the current elapsed time as its seed. If two **Random** objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two **Random** objects with the same seed, **3**:

```
Random generator1 = new Random(3);
System.out.print("From generator1: ");
for (int i = 0; i < 10; i++)
  System.out.print(generator1.nextInt(1000) + " ");

Random generator2 = new Random(3);
System.out.print("\nFrom generator2: ");
for (int i = 0; i < 10; i++)
  System.out.print(generator2.nextInt(1000) + " ");
```

The code generates the same sequence of random **int** values:

```
From generator1: 734 660 210 581 128 202 549 564 459 961
From generator2: 734 660 210 581 128 202 549 564 459 961
```

same sequence

**Note**

The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, often you need to reproduce the test cases from a fixed sequence of random numbers.

SecureRandom

**Note**

You can generate random numbers using the `java.security.SecureRandom` class rather than the `Random` class. The random numbers generated from the `Random` are deterministic and they can be predicted by hackers. The random numbers generated from the `SecureRandom` class are nondeterministic and are secure.

### 9.6.3 The `Point2D` Class

Java API has a convenient `Point2D` class in the `javafx.geometry` package for representing a point in a two-dimensional plane. The UML diagram for the class is shown in Figure 9.12.

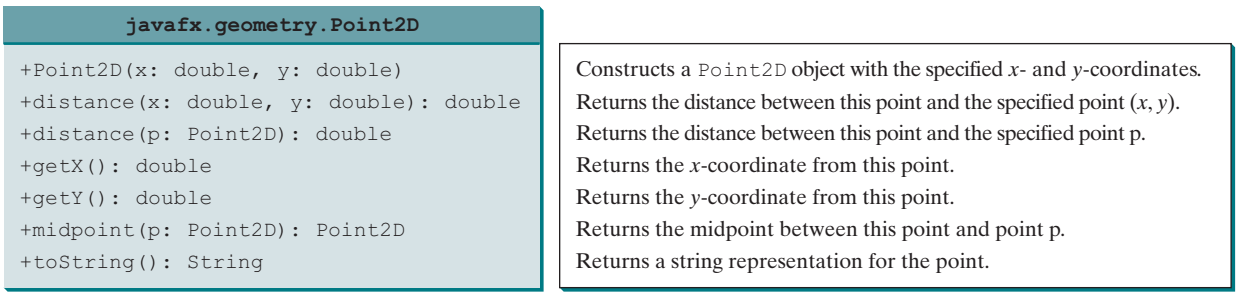



**FIGURE 9.12** A `Point2D` object represents a point with *x*- and *y*-coordinates.

You can create a `Point2D` object for a point with the specified *x*- and *y*-coordinates, use the `distance` method to compute the distance from this point to another point, and use the `toString()` method to return a string representation of the point. Listing 9.5 gives an example of using this class.

**LISTING 9.5** `TestPoint2D.java`

```
1  import java.util.Scanner;
2  import javafx.geometry.Point2D;
3
4  public class TestPoint2D {
5    public static void main(String[] args) {
6      Scanner input = new Scanner(System.in);
7
8      System.out.print("Enter point1's x-, y-coordinates: ");
9      double x1 = input.nextDouble();
10     double y1 = input.nextDouble();
11     System.out.print("Enter point2's x-, y-coordinates: ");
12     double x2 = input.nextDouble();
13     double y2 = input.nextDouble();
14
15     Point2D p1 = new Point2D(x1, y1);
16     Point2D p2 = new Point2D(x2, y2);
17     System.out.println("p1 is " + p1.toString());
18     System.out.println("p2 is " + p2.toString());
19     System.out.println("The distance between p1 and p2 is " +
20       p1.distance(p2));
21     System.out.println("The midpoint between p1 and p2 is " +
22       p1.midpoint(p2).toString());
23   }
24 }
```

create an object

invoke `toString()`

get distance

get midpoint

```
Enter point1's x-, y-coordinates: 1.5 5.5 Enter
Enter point2's x-, y-coordinates: -5.3 -4.4 Enter
p1 is Point2D [x = 1.5, y = 5.5]
p2 is Point2D [x = -5.3, y = -4.4]
The distance between p1 and p2 is 12.010412149464313
The midpoint between p1 and p2 is
Point2D [x = -1.9, y = 0.5499999999999998]
```

This program creates two objects of the `Point2D` class (lines 15 and 16). The `toString()` method returns a string that describes the object (lines 17 and 18). Invoking `p1.distance(p2)` returns the distance between the two points (line 20). Invoking `p1.midpoint(p2)` returns the midpoint between the two points (line 22).

**Note**

The `Point2D` class is defined in the `javafx.geometry` package, which is in the JavaFX module. To run this program, you need to install JavaFX. See Supplement II.F for installing and using JavaFX.

**9.6.1** How do you create a `Date` for the current time? How do you display the current time?

**9.6.2** How do you create a `Point2D`? Suppose `p1` and `p2` are two instances of `Point2D`, how do you obtain the distance between the two points? How do you obtain the midpoint between the two points?

**9.6.3** Which packages contain the classes `Date`, `Random`, `Point2D`, `System`, and `Math`?

## 9.7 Static Variables, Constants, and Methods

*A static variable is shared by all objects of the class. A static method cannot access instance members (i.e., instance data fields and methods) of the class.*

The data field `radius` in the `Circle` class 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:

Static vs. instance

instance variable

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

VideoNote

Static vs. instance

The `radius` in `circle1` is independent of the `radius` in `circle2` and is stored in a 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, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class.

static variable

static method

Let's modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created. When the first object of this class is created, `numberOfObjects` is `1`. When the second object is created, `numberOfObjects` becomes `2`. The UML of the new circle class is shown in Figure 9.13. The `Circle` class defines the instance variable `radius` and the static variable `numberOfObjects`, the instance methods `getRadius`, `setRadius`, and `getArea`, and the static method `getNumberOfObjects`. (Note static variables and methods are underlined in the UML class diagram.)

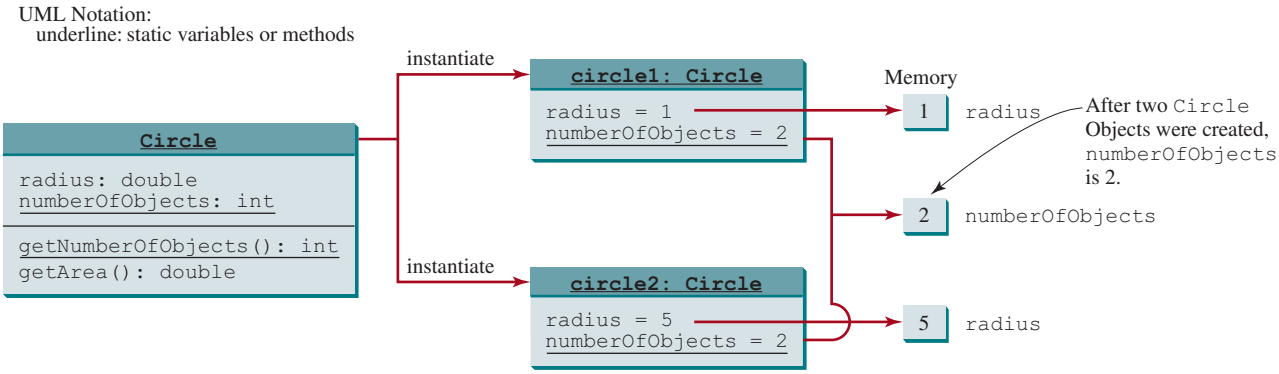



**FIGURE 9.13** Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

To declare a static variable or define a static method, put the modifier **static** in the variable or method declaration. The static variable **numberOfObjects** and the static method **getNumberOfObjects()** can be declared as follows:

declare static variable

```
static int numberOfObjects;
```

define static method

```
static int getNumberObjects() {
  return numberOfObjects;
}
```

declare constant

Constants in a class are shared by all objects of the class. Thus, constants should be declared as **final static**. For example, the constant **PI** in the **Math** class is defined as follows:

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

The new circle class is defined in Listing 9.6.

## LISTING 9.6 Circle.java (for CircleWithStaticMembers)

```
1  public class Circle {
2    /** The radius of the circle */
3    double radius;
4
5    /** The number of objects created */
6    static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    Circle() {
10     radius = 1;
11     numberOfObjects++;
12   }
13
14   /** Construct a circle with a specified radius */
15   Circle(double newRadius) {
16     radius = newRadius;
17     numberOfObjects++;
18   }
19
20   /** Return numberOfObjects */
21   static int getNumberOfObjects() {
22     return numberOfObjects;
23   }
24
25   /** Return the area of this circle */
```

static variable

increase by 1

increase by 1

static method

```
26    double getArea() {
27      return radius * radius * Math.PI;
28    }
29  }
```

Method **getNumberOfObjects()** in **Circle** is a static method. All the methods in the **Math** class are static. The **main** method is static, too.

Instance methods (e.g., **getArea()**) and instance data (e.g., **radius**) belong to instances and can be used only after the instances are created. They are accessed via a reference variable. Static methods (e.g., **getNumberOfObjects()**) and static data (e.g., **numberOfObjects**) can be accessed from a reference variable or from their class name.

The program in Listing 9.7 demonstrates how to use instance and static variables and methods and illustrates the effects of using them.

## LISTING 9.7 TestCircleWithStaticMembers.java

```
1  public class TestCircleWithStaticMembers {
2    /** Main method */
3    public static void main(String[] args) {
4      System.out.println("Before creating objects");
5      System.out.println("The number of Circle objects is " +
6        Circle.numberOfObjects);                                    static variable
7
8      // Create c1
9      Circle c1 = new Circle(); // Use the Circle class in Listing 9.6
10
11     // Display c1 BEFORE c2 is created
12     System.out.println("\nAfter creating c1");
13     System.out.println("c1: radius (" + c1.radius +              instance variable
14       ") and number of Circle objects (" +
15       c1.numberOfObjects + ")");                                  static variable
16
17     // Create c2
18     Circle c2 = new Circle(5);
19
20     // Modify c1
21     c1.radius = 9;                                               instance variable
22
23     // Display c1 and c2 AFTER c2 was created
24     System.out.println("\nAfter creating c2 and modifying c1");
25     System.out.println("c1: radius (" + c1.radius +
26       ") and number of Circle objects (" +
27       c1.numberOfObjects + ")");                                  static variable
28     System.out.println("c2: radius (" + c2.radius +
29       ") and number of Circle objects (" +
30       c2.numberOfObjects + ")");                                  static variable
31   }
32 }
```

```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

When you compile `TestCircleWithStaticMembers.java`, the Java compiler automatically compiles `Circle.java` if it has not been compiled since the last change.

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is `0`, since no objects have been created.

The `main` method creates two circles `c1` and `c2` (lines 9 and18). The instance variable `radius` in `c1` is modified to become `9` (line 21). This change does not affect the instance variable `radius` in `c2`, since these two instance variables are independent. The static variable `numberOfObjects` becomes `1` after `c1` is created (line 9), and it becomes `2` after `c2` is created (line 18).

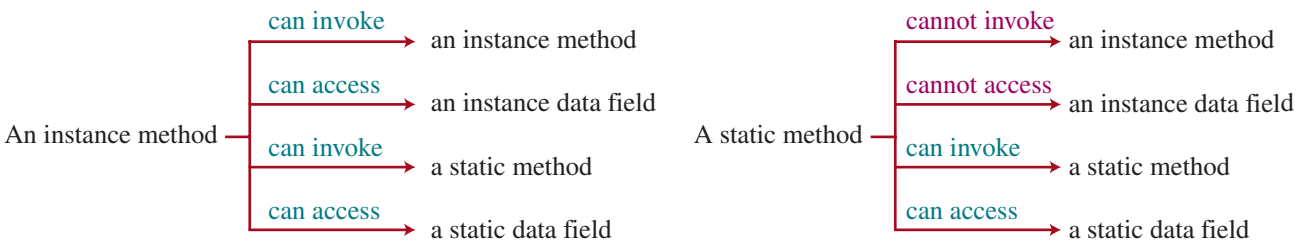
Note `PI` is a constant defined in `Math` and `Math.PI` references the constant. `c1.numberOfObjects` (line 27) and `c2.numberOfObjects` (line 30) are better replaced by `Circle.numberOfObjects`. This improves readability because other programmers can easily recognize the static variable. You can also replace `Circle.numberOfObjects` with `Circle.getNumberOfObjects()`.

use class name

**Tip**

Use `ClassName.methodName(arguments)` to invoke a static method and `ClassName.staticVariable` to access a static variable. This improves readability because this makes static methods and data easy to spot.

An instance method can invoke an instance or static method, and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don't belong to a particular object. The relationship between static and instance members is summarized in the following diagram:




For example, the following code is wrong.

```
1  public class A {
2    int i = 5;
3    static int  k = 2;
4
5    public static void main(String[] args) {
6      int j = i; // Wrong because i is an instance variable
7      m1(); // Wrong because m1() is an instance method
8    }
9
10   public void m1() {
11     // Correct since instance and static variables and methods
12     // can be used in an instance method
13     i = i + k + m2(i, k);
14   }
15
16   public static int m2(int i, int j) {
17     return (int)(Math.pow(i, j));
18   }
19 }
```

Note if you replace the preceding code with the following new code, the program would be fine, because the instance data field **i** and method **m1** are now accessed from an object **a** (lines 7 and 8):

```
1  public class A {
2    int i = 5;
3    static int k = 2;
4
5    public static void main(String[] args) {
6      A a = new A();
7      int j = a.i; // OK, a.i accesses the object's instance variable
8      a.m1(); // OK, a.m1() invokes the object's instance method
9    }
10
11   public void m1() {
12     i = i + k + m2(i, k);
13   }
14
15   public static int m2(int i, int j) {
16     return (int)(Math.pow(i, j));
17   }
18 }
```

### Design Guide

instance or static?

How do you decide whether a variable or a method should be instance or static? A variable or a method that is dependent on a specific instance of the class should be an instance variable or method. A variable or a method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, **radius** is an instance variable of the **Circle** class. Since the **getArea** method is dependent on a specific circle, it is an instance method. None of the methods in the **Math** class, such as **random**, **pow**, **sin**, and **cos**, is dependent on a specific instance. Therefore, these methods are static methods. The **main** method is static and can be invoked directly from a class.

### Caution

common design error

It is a common design error to define an instance method that should have been defined as static. For example, the method **factorial(int n)** should be defined as static, as shown next, because it is independent of any specific instance.

```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```
public class Test {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

**9.7.1** Suppose the class **F** is defined in (a). Let **f** be an instance of **F**. Which of the statements in (b) are correct?

```
public class F {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

(a)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

(b)

**9.7.2** Add the `static` keyword in the place of `?` if appropriate.

```
public class Test {
  int count;

  public ? void main(String[] args) {
    ...
  }

  public ? int getCount() {
    return count;
  }

  public ? int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

**9.7.3** Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

```
 1  public class C {
 2    Circle c = new Circle();
 3
 4    public static void main(String[] args) {
 5      method1();
 6    }
 7
 8    public void method1() {
 9      method2();
10    }
11
12    public static void method2() {
13      System.out.println("What is radius " + c.getRadius());
14    }
15  }
```


## 9.8 Visibility Modifiers

*Visibility modifiers can be used to specify the visibility of a class and its members.*

You can use the `public` visibility modifier for classes, methods, and data fields to denote they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.

package-private (or package-access)

**Note**

Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

using packages

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the *default package*.

Java recommends that you place classes into packages rather than using a default package. For simplicity, however, this book uses default packages. For more information on packages, see Supplement III.E, Packages.

In addition to the `public` and default visibility modifiers, Java provides the `private` and `protected` modifiers for class members. This section introduces the `private` modifier. The `protected` modifier will be introduced in Section 11.14, The `protected` Data and Methods.

The `private` modifier makes methods and data fields accessible only from within its own class. Figure 9.14 illustrates how a public, default, and private data field or method in class `C1` can be accessed from a class `C2` in the same package, and from a class `C3` in a different package.

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 c1 = new C1();
    can access c1.x;
    can access c1.y;
    cannot access c1.z;

    can invoke c1.m1();
    can invoke c1.m2();
    cannot invoke c1.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 c1 = new C1();
    can access c1.x;
    cannot access c1.y;
    cannot access c1.z;

    can invoke c1.m1();
    cannot invoke c1.m2();
    cannot invoke c1.m3();
  }
}
```

**FIGURE 9.14** The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

If a class is not defined as public, it can be accessed only within the same package. As shown in Figure 9.15, `C1` can be accessed from `C2`, but not from `C3`.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access p1.C1;
  can access p1.C2;
}
```

**FIGURE 9.15** A nonpublic class has package access.

The private modifier restricts private members from being accessed outside the class. However, there is no restriction on accessing members from inside the class. Therefore, objects instantiated in its own class can access its private members. As shown in Figure 9.16a, an object `c` of class `C` can access its private members, because `c` is defined inside its own class. However, in Figure 9.16b, an object `c` of class `C` cannot access its private members, because `c` is in the `Test` class.

inside access

```
public class C {
  private boolean x;

  public static void main(string[] args) {
    C c = new C();
    system.out.println(c.x);
    system.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

(a) This is okay because object c is used inside the class C.

```
public class Test {
  public static void main(string[] args) {
    C c = new C();
    system.out.println(c.x);
    system.out.println(c.convert());
  }
}
```

(b) This is wrong because x and convert are private in class C.

**FIGURE 9.16** An object can access its private members if it is defined in its own class.

**Caution**

The `private` modifier applies only to the members of a class. The `public` modifier can apply to a class or members of a class. Using the modifiers `public` and `private` on local variables would cause a compile error.

private constructor

**Note**

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a *private constructor*. For example, there is no reason to create an instance from the `Math` class, because all of its data fields and methods are static. To prevent the user from creating objects from the `Math` class, the constructor in `java.lang.Math` is defined as follows:

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

## 9.9 Data Field Encapsulation

*Making data fields private protects data and makes the class easy to maintain.*


Data field encapsulation

The data fields `radius` and `numberOfObjects` in the `Circle` class in Listing 9.6 can be modified directly (e.g., `c1.radius = 5` or `Circle.numberOfObjects = 10`). This is not a good practice—for two reasons:

VideoNote
Data field encapsulation

1. Data may be tampered with. For example, `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., `Circle.numberOfObjects = 10`).
2. The class becomes difficult to maintain and vulnerable to bugs. Suppose that you want to modify the `Circle` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle` class but also the programs that use it because the clients may have modified the radius directly (e.g., `c1.radius = -5`).

data field encapsulation

To prevent direct modifications of data fields, you should declare the data fields private, using the `private` modifier. This is known as *data field encapsulation*.

A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a *getter* method to return its value. To enable a private data field to be updated, provide a *setter* method to set a new value. A getter method is also referred to as an *accessor* and a setter to a *mutator*. A getter method has the following signature:

getter (or accessor)
setter (or mutator)

```
public returnType getPropertyName()
```

If the `returnType` is `boolean`, the getter method should be defined as follows by convention:

boolean accessor

```
public boolean isPropertyName()
```

A setter method has the following signature:

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

Let's create a new circle class with a private data-field radius and its associated accessor and mutator methods. The class diagram is shown in Figure 9.17. The new circle class is defined in Listing 9.8:

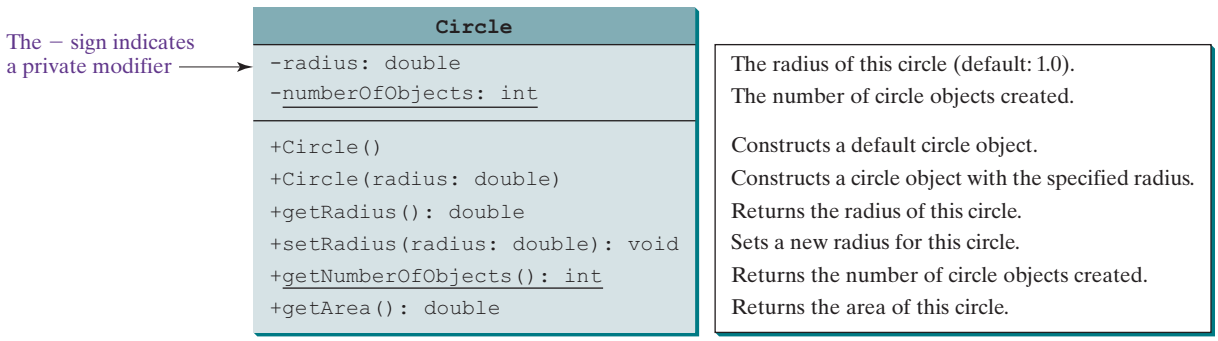



**FIGURE 9.17** The `Circle` class encapsulates circle properties and provides getter/setter and other methods.

## LISTING 9.8 Circle.java(for CircleWithPrivateDataFields)

```
1  public class Circle {
2    /** The radius of the circle */
3    private double radius = 1;                                  encapsulate radius
4
5    /** The number of objects created */
6    private static int numberOfObjects = 0;                     encapsulate
7                                                                  numberOfObjects
8    /** Construct a circle with radius 1 */
9    public Circle() {
10     numberOfObjects++;
11   }
12
13   /** Construct a circle with a specified radius */
14   public Circle(double newRadius) {
15     radius = newRadius;
16     numberOfObjects++;
```

```
17    }
18
19    /** Return radius */
20    public double getRadius() {
21      return radius;
22    }
23
24    /** Set a new radius */
25    public void setRadius(double newRadius) {
26      radius = (newRadius >= 0) ? newRadius : 0;
27    }
28
29    /** Return numberOfObjects */
30    public static int getNumberOfObjects() {
31      return numberOfObjects;
32    }
33
34    /** Return the area of this circle */
35    public double getArea() {
36      return radius * radius * Math.PI;
37    }
38  }
```

accessor method

mutator method

accessor method

The `getRadius()` method (lines 20–22) returns the radius and the `setRadius(newRadius)` method (lines 25–27) sets a new radius for the object. If the new radius is negative, `0` is set as the radius for the object. Since these methods are the only ways to read and modify the radius, you have total control over how the `radius` property is accessed. If you have to change the implementation of these methods, you don't need to change the client programs. This makes the class easy to maintain.

Listing 9.9 gives a client program that uses the `Circle` class to create a `Circle` object, and modifies the radius using the `setRadius` method.

## LISTING 9.9 TestCircleWithPrivateDataFields.java

```
1   public class TestCircleWithPrivateDataFields {
2     /** Main method */
3     public static void main(String[] args) {
4       // Create a circle with radius 5.0
5       Circle myCircle = new Circle(5.0);
6       System.out.println("The area of the circle of radius "
7         + myCircle.getRadius() + " is " + myCircle.getArea());
8
9       // Increase myCircle's radius by 10%
10      myCircle.setRadius(myCircle.getRadius() * 1.1);
11      System.out.println("The area of the circle of radius "
12        + myCircle.getRadius() + " is " + myCircle.getArea());
13
14      System.out.println("The number of objects created is "
15        + Circle.getNumberOfObjects());
16    }
17  }
```

invoke public method

invoke public method

invoke public method

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

Since `numberOfObjects` is private, it cannot be modified. This prevents tampering. For example, the user cannot set `numberOfObjects` to `100`. The only way to make it `100` is to create `100` objects of the `Circle` class.

Suppose you combined `TestCircleWithPrivateDataFields` and `Circle` into one class by moving the `main` method in `TestCircleWithPrivateDataFields` into `Circle`. Could you use `myCircle.radius` in the `main` method? See CheckPoint Question 9.9.3 for the answer.

**Design Guide**

To prevent data from being tampered with and to make the class easy to maintain, declare data fields private.

**Note**

From now on, all data fields should be declared private, and all constructors and methods should be defined public, unless specified otherwise.


**9.9.1** What is an accessor method? What is a mutator method? What are the naming conventions for accessor methods and mutator methods?

**9.9.2** What are the benefits of data field encapsulation?

**9.9.3** In the following code, `radius` is private in the `Circle` class, and `myCircle` is an object of the `Circle` class. Does the highlighted code cause any problems? If so, explain why.

```
public class Circle {
  private double radius = 1;

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

  public static void main(String[] args) {
    Circle myCircle = new Circle();
    System.out.println("Radius is " + myCircle.radius);
  }
}
```

## 9.10 Passing Objects to Methods

*Passing an object to a method is to pass the reference of the object.*


You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

```
1  public class Test {
2    public static void main(String[] args) {
3      // Circle is defined in Listing 9.8
4      Circle myCircle = new Circle(5.0);
5      printCircle(myCircle);
6    }
7
8    public static void printCircle(Circle c) {
9      System.out.println("The area of the circle of radius "
10       + c.getRadius() + " is " + c.getArea());
11   }
12 }
```

pass an object

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.

pass-by-value

The program in Listing 9.10 demonstrates the difference between passing a primitive-type value and passing a reference value.

## LISTING 9.10 TestPassObject.java

```
1  public class TestPassObject {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a Circle object with radius 1
5      Circle myCircle =
6        new Circle(1); // Use the Circle class in Listing 9.8
7
8      // Print areas for radius 1, 2, 3, 4, and 5.
9      int n = 5;
10     printAreas(myCircle, n);
11
12     // See myCircle.radius and times
13     System.out.println("\n" + "Radius is " + myCircle.getRadius());
14     System.out.println("n is " + n);
15   }
16
17   /** Print a table of areas for radius */
18   public static void printAreas(Circle c, int times) {
19     System.out.println("Radius \t\tArea");
20     while (times >= 1) {
21       System.out.println(c.getRadius() + "\t\t" + c.getArea());
22       c.setRadius(c.getRadius() + 1);
23       times--;
24     }
25   }
26 }
```

pass object

object parameter

```
Radius        Area
1.0           3.141592653589793
2.0           12.566370614359172
3.0           28.274333882308138
4.0           50.26548245743669
5.0           78.53981633974483
Radius is 6.0
n is 5
```

The **Circle** class is defined in Listing 9.8. The program passes a **Circle** object **myCircle** and an integer value from **n** to invoke **printAreas(myCircle, n)** (line 10), which prints a table of areas for radii **1**, **2**, **3**, **4**, and **5**, as presented in the sample output.

Figure 9.18 shows the call stack for executing the methods in the program. Note the objects are stored in a heap (see Section 7.6).

When passing an argument of a primitive data type, the value of the argument 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 an argument 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 the properties of the object through **c** inside the **printAreas** method has the same effect as doing so outside the method through the variable **myCircle**. Pass-by-value on references can be best described semantically as *pass-by-sharing*; that is, the object referenced in the method is the same as the object being passed.

pass-by-sharing

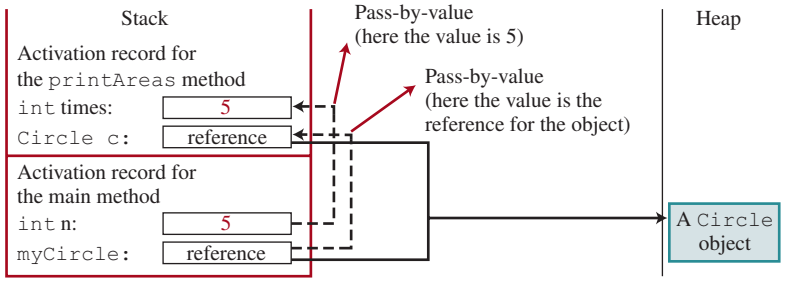



**FIGURE 9.18** The value of `n` is passed to `times`, and the reference to `myCircle` is passed to `c` in the `printAreas` method.

**9.10.1** 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 programs:


```
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++;
  }
}
```

```
public class Count {
  public int count;

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

  public Count () {
    count = 1;
  }
}
```

**9.10.2** 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);

    swap1(circle1, circle2);
    System.out.println("After swap1: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);

    swap2(circle1, circle2);
    System.out.println("After swap2: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);
  }

  public static void swap1(Circle x, Circle y) {
    Circle temp = x;
    x = y;
    y = temp;
  }
```

```
  public static void swap2(Circle x, Circle y) {
    double temp = x.radius;
    x.radius = y.radius;
    y.radius = temp;
  }
}

class Circle {
  double radius;

  Circle(double newRadius) {
    radius = newRadius;
  }
}
```

**9.10.3** Show the output of the following code:

```
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;
  }
}
```

(a)

```
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;
  }
}
```

(b)

```
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;
}
```

(c)

```
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;
  }
}
```

(d)

**9.10.4** What is the output of the following programs?

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = null;
    m1(date);
    System.out.println(date);
  }

  public static void m1(Date date) {
    date = new Date();
  }
}
```

(a)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = new Date(7654321);
  }
}
```

(b)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date.setTime(7654321);
  }
}
```

(c)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = null;
  }
}
```

(d)

## 9.11 Array of Objects

*An array can hold objects as well as primitive-type values.*


Chapter 7, Single-Dimensional Arrays, described how to create arrays of primitive-type elements. 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 `circleArray`, you can use a `for` loop as follows:

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

An array of objects is actually an *array of reference variables*. Thus, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 9.19. `circleArray` references the entire array, and `circleArray[1]` references a `Circle` object.

**Note**

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

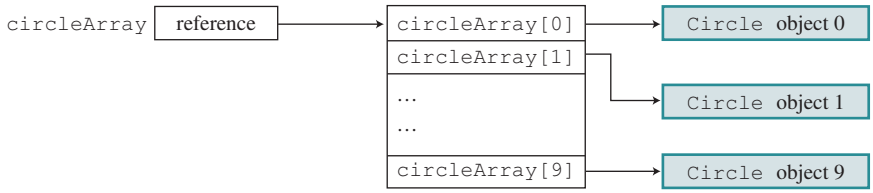



**FIGURE 9.19** In an array of objects, an element of the array contains a reference to an object.

Listing 9.11 gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates `circleArray`, an array composed of five `Circle` objects; it then initializes circle radii with random values and displays the total area of the circles in the array.

## LISTING 9.11 TotalArea.java

array of objects

return array of objects

pass array of objects

```
1  public class TotalArea {
2    /** Main method */
3    public static void main(String[] args) {
4      // Declare circleArray
5      Circle[] circleArray;
6
7      // Create circleArray
8      circleArray = createCircleArray();
9
10     // Print circleArray and total areas of the circles
11     printCircleArray(circleArray);
12   }
13
14   /** Create an array of Circle objects */
15   public static Circle[] createCircleArray() {
16     Circle[] circleArray = new Circle[5];
17
18     for (int i = 0; i < circleArray.length; i++) {
19       circleArray[i] = new Circle(Math.random() * 100);
20     }
21
22     // Return Circle array
23     return circleArray;
24   }
25
26   /** Print an array of circles and their total area */
27   public static void printCircleArray(Circle[] circleArray) {
28     System.out.printf("%-30s%-15s\n", "Radius", "Area");
29     for (int i = 0; i < circleArray.length; i++) {
30       System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
31         circleArray[i].getArea());
32     }
33
34     System.out.println("--------------------------------");
35
36     // Compute and display the result
37     System.out.printf("%-30s%-15f\n", "The total area of circles is",
38       sum(circleArray));
39   }
40
```

```
41    /** Add circle areas */
42    public static double sum(Circle[] circleArray) {
43      // Initialize sum
44      double sum = 0;
45
46      // Add areas to sum
47      for (int i = 0; i < circleArray.length; i++)
48        sum += circleArray[i].getArea();
49
50      return sum;
51    }
52  }
```

pass array of objects

```
Radius                        Area
70.577708                     15649.941866
44.152266                      6124.291736
24.867853                      1942.792644
 5.680718                       101.380949
36.734246                      4239.280350
--------------------------------------------------

The total area of circles is 28056.687544
```

The program invokes `createCircleArray()` (line 8) to create an array of five circle objects. Several circle classes were introduced in this chapter. This example uses the `Circle` class introduced in Section 9.9, Data Field Encapsulation.

The circle radii are randomly generated using the `Math.random()` method (line 19). The `createCircleArray` method returns an array of `Circle` objects (line 23). The array is passed to the `printCircleArray` method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed by invoking the `sum` method (line 38), which takes the array of `Circle` objects as the argument and returns a `double` value for the total area.

**9.11.1** What is wrong in the following code?

```
1  public class Test {
2    public static void main(String[] args) {
3      java.util.Date[] dates = new java.util.Date[10];
4      System.out.println(dates[0]);
5      System.out.println(dates[0].toString());
6    }
7  }
```

## 9.12 Immutable Objects and Classes

*You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.*


VideoNote
Immutable objects and this keyword

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as *immutable object* and its class as *immutable class*. The `String` class, for example, is immutable. If you deleted the setter method in the `Circle` class in Listing 9.8, the class would be immutable because radius is private and cannot be changed without a setter method.

immutable object
immutable class

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A class with all private data fields and no mutators is not

Student class

necessarily immutable. For example, the following **Student** class has all private data fields and no setter methods, but it is not an immutable class:

```
1  public class Student {
2    private int id;
3    private String name;
4    private java.util.Date dateCreated;
5
6    public Student(int ssn, String newName) {
7      id = ssn;
8      name = newName;
9      dateCreated = new java.util.Date();
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public String getName() {
17     return name;
18   }
19
20   public java.util.Date getDateCreated() {
21     return dateCreated;
22   }
23 }
```

As shown in the following code, the data field **dateCreated** is returned using the **getDateCreated()** method. This is a reference to a **Date** object. Through this reference, the content for **dateCreated** can be changed.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, "John");
    java.util.Date dateCreated = student.getDateCreated();
    dateCreated.setTime(200000); // Now dateCreated field is changed!
  }
}
```

For a class to be immutable, it must meet the following requirements:

- All data fields must be private.
- There can't be any mutator methods for data fields.
- No accessor methods can return a reference to a data field that is mutable.

Interested readers may refer to Supplement III.U for an extended discussion on immutable objects.


**9.12.1** If a class contains only private data fields and no setter methods, is the class immutable?

**9.12.2** If all the data fields in a class are private and of primitive types, and the class doesn't contain any setter methods, is the class immutable?

**9.12.3** Is the following class immutable?

```
public class A {
  private int[] values;

  public int[] getValues() {
```

```
      return values;
    }
  }
```

## 9.13 The Scope of Variables

*The scope of instance and static variables is the entire class, regardless of where the variables are declared.*


Section 6.9 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 the variables in the context of a class.

Instance and static variables in a class are referred to as the *class's variables* or *data fields*. A variable defined inside a method is referred to as a *local variable*. The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class, as shown in Figure 9.20a. The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first, as shown in Figure 9.20b. For consistency, this book declares data fields at the beginning of the class.

class's variables

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

  private double radius = 1;
}
```

(a)

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

(b)

(a) The variable `radius` and method `getArea()` can be declared in any order.

(b) `i` has to be declared before `j` because `j`'s initial value is dependent on `i`.

**FIGURE 9.20** Members of a class can be declared in any order, with one exception.

You can declare a class's variable only once, but you can declare the same variable name in a method many times in different nonnesting 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 both as an instance variable and as a local variable in the method:

hidden variables

```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

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

What is the output for `f.p()`, where `f` is an instance of `F`? The output for `f.p()` is `1` for `x` and `0` for `y`. Here is why:

- `x` is declared as a data field with the initial value of `0` in the class, but it is also declared 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 `y` is accessible inside the method.

**Tip**

To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters. We will discuss hidden data fields by method parameters in the next section.

**9.13.1** What is the output of the following program?

```
public class Test {
  private static int i = 0;
  private 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);
  }
}
```

## 9.14 The `this` Reference

*The keyword* **`this`** *refers to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.*

`this` keyword

When an instance method is called on an object, the **`this`** *keyword* is set to this object. You can use the **`this`** *keyword* to reference the object's instance members in the class. For example, the following code in (a) uses **`this`** to reference the object's **`radius`** and invokes its **`getArea()`** method explicitly. Some instructors prefer using the **`this`** keyword explicitly in the code, because it clearly distinguishes the instance variables from local variables. However, the **`this`** reference is normally omitted for brevity as shown in (b). Nevertheless, the **`this`** keyword is needed to reference a data field hidden by a method or constructor parameter, or to invoke an overloaded constructor.

VideoNote

The this keyword

```
public class Circle {
  private double radius;

  ...

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

  public String toString() {
    return "radius: " + this.radius
      + "area: " + this.getArea();
  }
}
```

(a)

Equivalent

```
public class Circle {
  private double radius;

  ...

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

  public String toString() {
    return "radius: " + radius
      + "area: " + getArea();
  }
}
```

(b)

## 9.14.1 Using this to Reference Data Fields

It is a good practice to use the data field as the parameter name in a setter method or a constructor to make the code easy to read and to avoid creating unnecessary names. In this case, you need to use the `this` keyword to reference the data field in the setter method. For example, the `setRadius` method can be implemented as shown in (a). It would be wrong if it is implemented as shown in (b).

Refers to data field `radius` in this object.

```
private double radius;

public void setRadius(double radius) {
  this.radius = radius;
}
```

(a) Refers to data field radius in this object

Here, `radius` is the parameter in the method.

```
private double radius = 1;

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

(b) Refers to parameter radius in the method.

The data field `radius` is hidden by the parameter `radius` in the setter method. You need to reference the data field name in the method using the syntax `this.radius`. 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 Figure 9.21a.

reference the data field

```
public class F {
  private int i = 5;
  private static double k = 0;

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

  public static
      void setK(double k) {
    F.k = k;

  }
  // other methods omitted
}
```

(a)

Suppose that f1 and f2 are two objects of F created as follows:.

```
F f1 = new F();
F f2 = new F();
```

Invoking `f1.setI(10)` is to execute `this.i = 10`, where this is an alias for `f1`

Invoking `f2.setI(45)` is to execute `this.i = 45`, where this is an alias for `f2`

Invoking `f2.setK(33)` is to execute `F.k = 33`. `setK` is a static method

(b)

**FIGURE 9.21** The keyword `this` refers to the calling object that invokes the method.

The `this` keyword gives us a way to reference the object that invokes an instance method. To invoke `f1.setI(10)`, `this.i = i` is executed, which assigns the value of parameter `i` to the data field `i` of this calling object `f1`. The keyword `this` is an alias for `f1`, as shown in Figure 9.21b. The line `F.k = k` means the value in parameter `k` is assigned to the static data field `k` of the class, which is shared by all the objects of the class.

### 9.14.2 Using `this` to Invoke a Constructor

The `this` keyword can be used to invoke another constructor of the same class. For example, you can rewrite the `Circle` class as follows:

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

  public Circle() {
    this(1.0);
  }

  ...
}
```

The `this` keyword is used to reference the data field radius of the object being constructed.

The `this` keyword is used to invoke another constructor.

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

**Note**

Java requires that the `this(arg-list)` statement appear first in the constructor before any other executable statements.

**Tip**

If a class has multiple constructors, it is better to implement them using `this(arg-list)` as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using `this(arg-list)`. This syntax often simplifies coding and makes the class easier to read and to maintain.


**9.14.1** Describe the role of the `this` keyword.

**9.14.2** What is wrong in the following code?

```
1  public class C {
2    private int p;
3
4    public C() {
5      System.out.println("C's no-arg constructor invoked");
6      this(0);
7    }
8
9    public C(int p) {
10     p = p;
11   }
12
13   public void setP(int p) {
14     p = p;
15   }
16 }
```

**9.14.3** What is wrong in the following code?

```
public class Test {
  private int id;

  public void m1() {
    this.id = 45;
  }
```

```
  public void m2() {
    Test.id = 45;
  }
}
```

## Key Terms

accessor 347
action 324
anonymous object 333
attribute 324
behavior 324
class 324
class's variable 357
client 327
constructor 324
data field 324
data field encapsulation 346
default constructor 331
dot operator (`.`) 333
getter 347
immutable class 355
immutable object 355
instance 324
instance method 333
instance variable 333
instantiation 324
mutator 347
no-arg constructor 327
`null` value 334
object 324
object-oriented programming (OOP) 324
package-private (or package-access) 344
private constructor 346
property 324
public class 327
reference type 332
reference variable 332
setter 347
state 324
static method 339
static variable 339
`this` keyword 358
Unified Modeling Language (UML) 325

## Chapter Summary

1. A *class* is a template for *objects*. It defines the *properties* of objects and provides *constructors* for creating objects and methods for manipulating them.

2. A class is also a data type. You can use it to declare object *reference variables*. 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 most of the time the distinction can be ignored.

3. An object is an *instance* of a class. You use the **`new`** operator to create an object and the *dot operator* (`.`) to access members of that object through its reference variable.

4. An *instance variable* or *method* 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. A *static method* is a method that can be invoked without using instances.

5. Every instance of a class can access the class's static variables and methods. For clarity, however, it is better to invoke static variables and methods using **`ClassName.variable`** and **`ClassName.method`**.

6. Visibility modifiers specify how the class, method, and data are accessed. A **`public`** class, method, or data is accessible to all clients. A **`private`** method or data is accessible only inside the class.

7. You can provide a getter (accessor) method or a setter (mutator) method to enable clients to see or modify the data.

8. A getter method has the signature `public returnType getPropertyName()`. If the `returnType` is `boolean`, the getter method should be defined as `public boolean isPropertyName()`. A setter method has the signature `public void setPropertyName(dataType propertyValue)`.

9. All 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.

10. A Java array is an object that can contain primitive-type values or object-type values. When an array of objects is created, its elements are assigned the default value of `null`.

11. Once it is created, an *immutable object* cannot be modified. To prevent users from modifying an object, you can define *immutable classes*.

12. The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables can be declared anywhere in the class. For consistency, they are declared at the beginning of the class in this book.

13. The keyword `this` can be used to refer to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.


## Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

## Programming Exercises

three objectives

### Pedagogical Note

The exercises in Chapters 9–13 help you to achieve three objectives:

1. Design classes and draw UML class diagrams.
2. Implement classes from the UML.
3. Use classes to develop applications.

Students can download solutions for the UML diagrams for the even-numbered exercises from the Companion Website and instructors can download all solutions from the Instructor Website.

Starting from Section 9.7, all data fields should be declared private and all constructors and methods should be defined public unless specified otherwise.

### Sections 9.2–9.5

**9.1** (*The `Rectangle` class*) Following the example of the `Circle` class in Section 9.2, design a class named `Rectangle` to represent a rectangle. The class contains:

- Two `double` data fields named `width` and `height` that specify the width and height of the rectangle. The default values are `1` for both `width` and `height`.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified `width` and `height`.
- A method named `getArea()` that returns the area of this rectangle.
- A method named `getPerimeter()` that returns the perimeter.

Draw the UML diagram for the class then implement the class. Write a test program that creates two `Rectangle` objects—one with width `4` and height `40`, and

the other with width 3.5 and height 35.9. Display the width, height, area, and perimeter of each rectangle in this order.

**9.2** (*The Stock class*) Following the example of the Circle class in Section 9.2, design a class named Stock that contains:

- A string data field named symbol for the stock's symbol.
- A string data field named name for the stock's name.
- A double data field named previousClosingPrice that stores the stock price for the previous day.
- A double data field named currentPrice that stores the stock price for the current time.
- A constructor that creates a stock with the specified symbol and name.
- A method named getChangePercent() that returns the percentage changed from previousClosingPrice to currentPrice.

Draw the UML diagram for the class then implement the class. Write a test program that creates a Stock object with the stock symbol ORCL, the name Oracle Corporation, and the previous closing price of 34.5. Set a new current price to 34.35 and display the price-change percentage.

## Section 9.6

***9.3** (*Use the Date class*) Write a program that creates a Date object, sets its elapsed time to 10000, 100000, 1000000, 10000000, 100000000, 1000000000, 10000000000, and 100000000000, and displays the date and time using the toString() method, respectively.

***9.4** (*Use the Random class*) Write a program that creates a Random object with seed 1000 and displays the first 50 random integers between 0 and 100 using the nextInt(100) method.

***9.5** (*Use the GregorianCalendar class*) Java API has the GregorianCalendar class in the java.util package, which you can use to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods get(GregorianCalendar.YEAR), get(GregorianCalendar.MONTH), and get(GregorianCalendar.DAY_OF_MONTH) return the year, month, and day. Write a program to perform two tasks:

1. Display the current year, month, and day.
2. The GregorianCalendar class has the setTimeInMillis(long), which can be used to set a specified elapsed time since January 1, 1970. Set the value to 1234567898765L and display the year, month, and day.

## Sections 9.7–9.9

***9.6** (*Stopwatch*) Design a class named StopWatch. The class contains:

- Private data fields startTime and endTime with getter methods.
- A no-arg constructor that initializes startTime with the current time.
- A method named start() that resets the startTime to the current time.
- A method named stop() that sets the endTime to the current time.
- A method named getElapsedTime() that returns the elapsed time for the stopwatch in milliseconds.

Draw the UML diagram for the class then implement the class. Write a test program that measures the execution time of sorting 100,000 numbers using selection sort.

**9.7** (*The Account class*) Design a class named Account that contains:

- A private int data field named id for the account (default 0).
- A private double data field named balance for the account (default 0).

- A private `double` data field named `annualInterestRate` that stores the current interest rate (default `0`). Assume that all accounts have the same interest rate.
- A private `Date` data field named `dateCreated` that stores the date when the account was created.
- A no-arg constructor that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for `id`, `balance`, and `annualInterestRate`.
- The accessor method for `dateCreated`.
- A method named `getMonthlyInterestRate()` that returns the monthly interest rate.
- A method named `getMonthlyInterest()` that returns the monthly interest.
- A method named `withdraw` that withdraws a specified amount from the account.
- A method named `deposit` that deposits a specified amount to the account.

Draw the UML diagram for the class then implement the class. (*Hint*: The method `getMonthlyInterest()` is to return monthly interest, not the interest rate. Monthly interest is `balance * monthlyInterestRate`. `monthlyInterestRate` is `annualInterestRate / 12`. Note `annualInterestRate` is a percentage, for example 4.5%. You need to divide it by 100.)

Write a test program that creates an `Account` object with an account ID of 1122, a balance of $20,000, and an annual interest rate of 4.5%. Use the `withdraw` method to withdraw $2,500, use the `deposit` method to deposit $3,000, and print the balance, the monthly interest, and the date when this account was created.

VideoNote
The Fan class

**9.8** (*The Fan class*) Design a class named `Fan` to represent a fan. The class contains:

- Three constants named `SLOW`, `MEDIUM`, and `FAST` with the values `1`, `2`, and `3` to denote the fan speed.
- A private `int` data field named `speed` that specifies the speed of the fan (the default is `SLOW`).
- A private `boolean` data field named `on` that specifies whether the fan is on (the default is `false`).
- A private `double` data field named `radius` that specifies the radius of the fan (the default is `5`).
- A string data field named `color` that specifies the color of the fan (the default is `blue`).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named `toString()` that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns the fan color and radius along with the string "fan is off" in one combined string.

Draw the UML diagram for the class then implement the class. Write a test program that creates two `Fan` objects. Assign maximum speed, radius `10`, color `yellow`, and turn it on to the first object. Assign medium speed, radius `5`, color `blue`, and turn it off to the second object. Display the objects by invoking their `toString` method.

****9.9** (*Geometry: n-sided regular polygon*) In an *n*-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named `RegularPolygon` that contains:

- A private `int` data field named `n` that defines the number of sides in the polygon with default value `3`.

- A private `double` data field named `side` that stores the length of the side with default value `1`.
- A private `double` data field named `x` that defines the *x*-coordinate of the polygon's center with default value `0`.
- A private `double` data field named `y` that defines the *y*-coordinate of the polygon's center with default value `0`.
- A no-arg constructor that creates a regular polygon with default values.
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (`0`, `0`).
- A constructor that creates a regular polygon with the specified number of sides, length of side, and *x*- and *y*-coordinates.
- The accessor and mutator methods for all data fields.
- The method `getPerimeter()` that returns the perimeter of the polygon.
- The method `getArea()` that returns the area of the polygon. The formula for computing the area of a regular polygon is

$$\text{Area} = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}.$$

Draw the UML diagram for the class then implement the class. Write a test program that creates three `RegularPolygon` objects, created using the no-arg constructor, using `RegularPolygon(6, 4)`, and using `RegularPolygon(10, 4, 5.6, 7.8)`. For each object, display its perimeter and area.

***9.10** (*Algebra: quadratic equations*) Design a class named `QuadraticEquation` for a quadratic equation $ax^2 + bx + c = 0$. The class contains:

- Private data fields `a`, `b`, and `c` that represent three coefficients.
- A constructor with the arguments for `a`, `b`, and `c`.
- Three getter methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning two roots of the equation

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

Draw the UML diagram for the class then implement the class. Write a test program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is 0, display the one root. Otherwise, display "The equation has no roots." See Programming Exercise 3.1 for sample runs.

***9.11** (*Algebra: 2 × 2 linear equations*) Design a class named `LinearEquation` for a 2 × 2 system of linear equations:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \quad x = \frac{ed - bf}{ad - bc} \quad y = \frac{af - ec}{ad - bc}$$

The class contains:

- Private data fields `a`, `b`, `c`, `d`, `e`, and `f`.
- A constructor with the arguments for `a`, `b`, `c`, `d`, `e`, and `f`.
- Six getter methods for `a`, `b`, `c`, `d`, `e`, and `f`.
- A method named `isSolvable()` that returns true if $ad - bc$ is not 0.
- Methods `getX()` and `getY()` that return the solution for the equation.

Draw the UML diagram for the class then implement the class. Write a test program that prompts the user to enter `a`, `b`, `c`, `d`, `e`, and `f` and displays the result. If $ad - bc$ is 0, report that "The equation has no solution." See Programming Exercise 3.3 for sample runs.

**9.12 (*Geometry: intersecting point*) Suppose two line segments intersect. The two endpoints for the first line segment are (`x1`, `y1`) and (`x2`, `y2`) and for the second line segment are (`x3`, `y3`) and (`x4`, `y4`). Write a program that prompts the user to enter these four endpoints and displays the intersecting point. As discussed in Programming Exercise 3.25, the intersecting point can be found by solving a linear equation. Use the `LinearEquation` class in Programming Exercise 9.11 to solve this equation. See Programming Exercise 3.25 for sample runs.

**9.13 (*The `Location` class*) Design a class named `Location` for locating a maximal value and its location in a two-dimensional array. The class contains public data fields `row`, `column`, and `maxValue` that store the maximal value and its indices in a two-dimensional array with `row` and `column` as `int` types and `maxValue` as a `double` type.

Write the following method that returns the location of the largest element in a two-dimensional array:

```
public static Location locateLargest(double[][] a)
```

The return value is an instance of `Location`. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns in the array: 3 4 Enter
Enter the array:
23.5 35 2 10 Enter
4.5 3 45 3.5 Enter
35 44 5.5 9.6 Enter
The location of the largest element is 45 at (1, 2)
```