

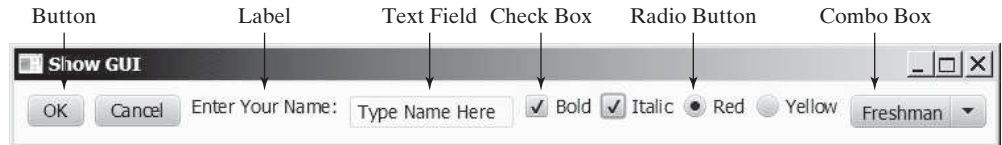
## 9.1 Introduction



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

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.*

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.

- 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 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.

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.

A Java class uses variables to define data fields and methods to define actions. Additionally, 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.



### VideoNote

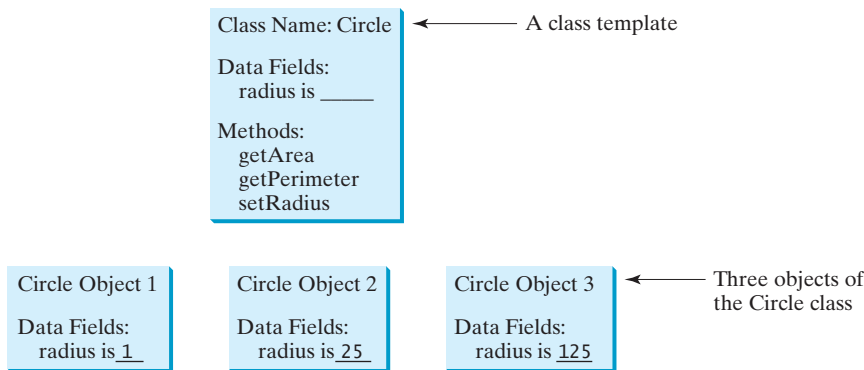
Define classes and objects

object  
state of an object  
properties  
attributes  
data fields  
behavior  
actions

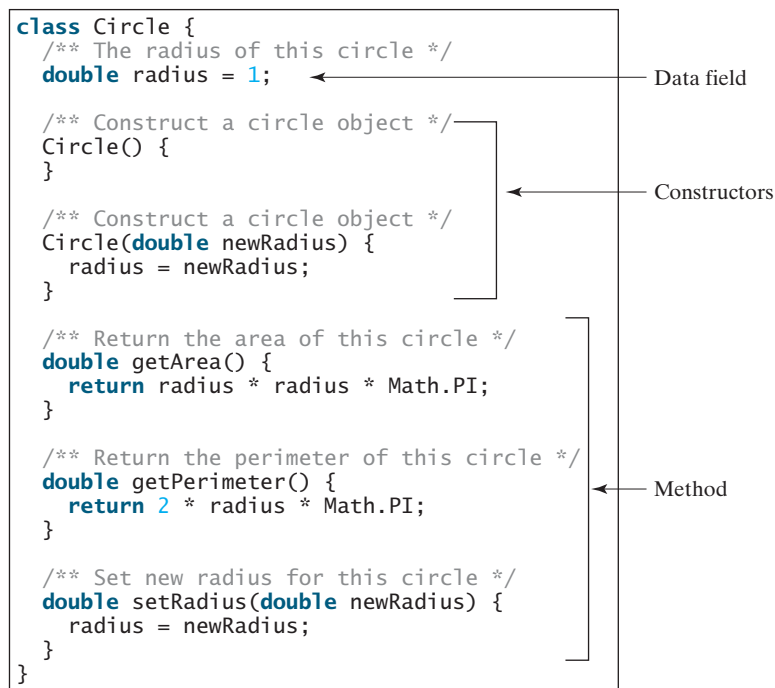
class  
contract

instantiation  
instance

data field  
method  
constructors



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



**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*.

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

dataFieldName: dataType

The constructor is denoted as

ClassName(parameterName: parameterType)

main class  
Unified Modeling Language  
(UML)  
class diagram

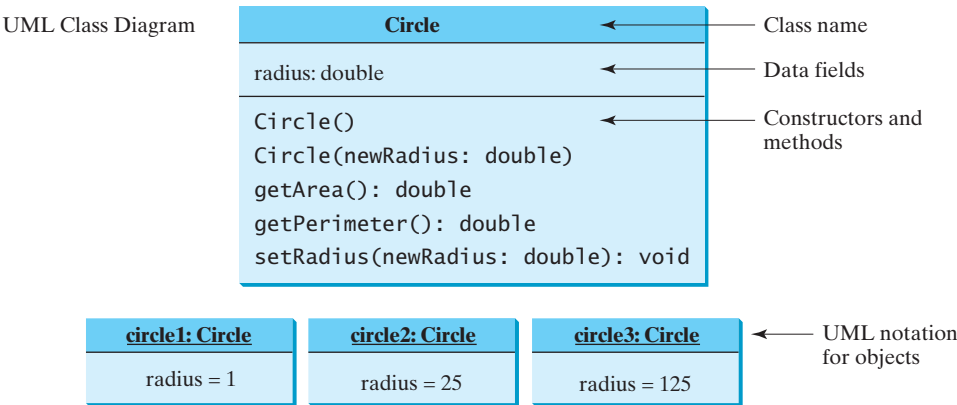


FIGURE 9.4 Classes and objects can be represented using UML notation.

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.



#### Note

To avoid a naming conflict with several enhanced versions of the **Circle** class introduced later in the chapter, the **Circle** class in this example is named **SimpleCircle**. For simplicity, we will still refer to the class in the text as **Circle**.

avoid naming conflicts

#### LISTING 9.1 TestSimpleCircle.java

main class  
main method  
create object  
create object  
create object

```
1 public class TestSimpleCircle {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a circle with radius 1
5         SimpleCircle circle1 = new SimpleCircle();
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        SimpleCircle circle2 = new SimpleCircle(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        SimpleCircle circle3 = new SimpleCircle(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 SimpleCircle {
28      double radius;
29
30      /** Construct a circle with radius 1 */
31      SimpleCircle() {
32          radius = 1;
33      }
34
35      /** Construct a circle with a specified radius */
36      SimpleCircle(double newRadius) {
37          radius = newRadius;
38      }
39
40      /** Return the area of this circle */
41      double getArea() {
42          return radius * radius * Math.PI;
43      }
44
45      /** Return the perimeter of this circle */
46      double getPerimeter() {
47          return 2 * radius * Math.PI;
48      }
49
50      /** Set a new radius for this circle */
51      void setRadius(double newRadius) {
52          radius = newRadius;
53      }
54  }

```

class SimpleCircle  
data field

no-arg constructor

second constructor

getArea

getPerimeter

setRadius

```

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

```

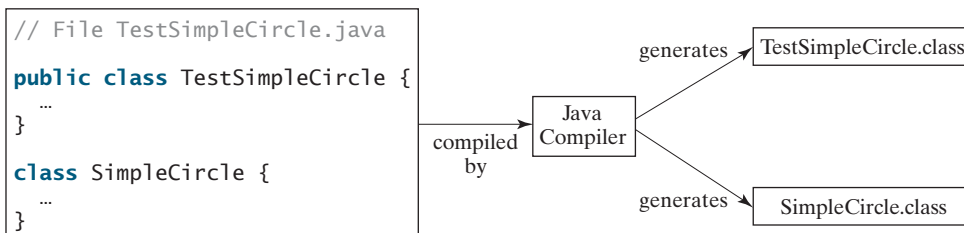


The program contains two classes. The first of these, `TestSimpleCircle`, is the main class. Its sole purpose is to test the second class, `SimpleCircle`. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the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

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 `TestSimpleCircle.java`, since `TestSimpleCircle` is public. Each class in the source code is compiled into a `.class` file. When you compile `TestSimpleCircle.java`, two class files `TestSimpleCircle.class` and `SimpleCircle.class` are generated, as shown in Figure 9.5.

client

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 SimpleCircle()` creates an object with radius `1` (line 5), `new SimpleCircle(25)` creates an object with radius `25` (line 10), and `new SimpleCircle(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 21–22.

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

### LISTING 9.2 SimpleCircle.java

```

1  public class SimpleCircle {
2      /** Main method */
3      public static void main(String[] args) {
4          // Create a circle with radius 1
5          SimpleCircle circle1 = new SimpleCircle();
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         SimpleCircle circle2 = new SimpleCircle(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         SimpleCircle circle3 = new SimpleCircle(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     SimpleCircle() {
29         radius = 1;
30     }
31
32     /** Construct a circle with a specified radius */
33     SimpleCircle(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, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 9.6.

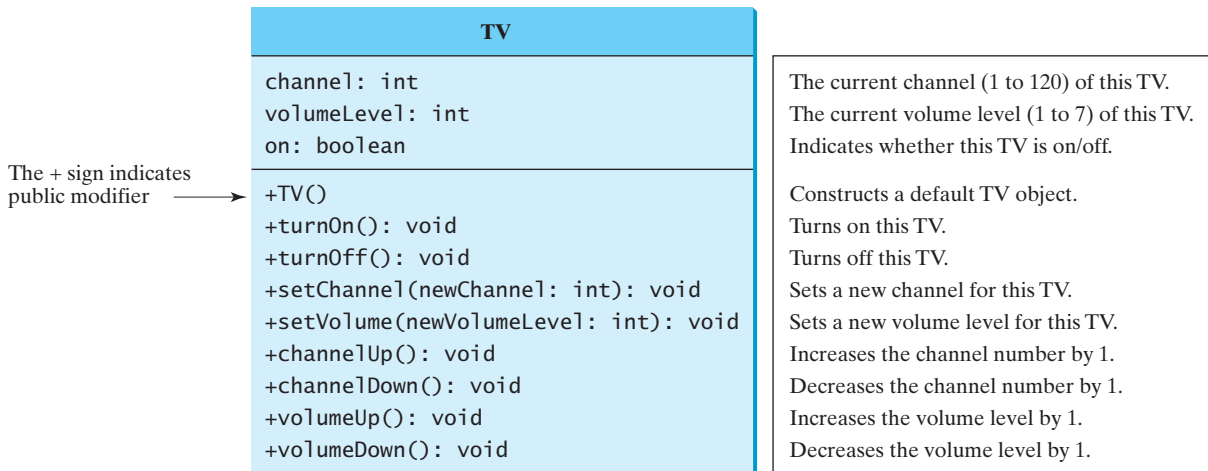


FIGURE 9.6 The TV class models TV sets.

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

```

data fields

constructor

turn on TV

turn off TV

```

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

```

The constructor and methods in the **TV** class are defined public so they can be accessed from other classes. Note that 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 that 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 }

```

```
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 follow discuss these issues in detail.

- 9.1** Describe the relationship between an object and its defining class.
- 9.2** How do you define a class?
- 9.3** How do you declare an object's reference variable?
- 9.4** 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.
- Constructors do not have a return type—not even **void**.
- Constructors are invoked using the **new** operator when an object is created. Constructors play the role of initializing objects.

constructor's name

no return type

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,

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

no void

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

- 9.5** What are the differences between constructors and methods?
- 9.6** When will a class have a default constructor?





## 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.

### 9.5.1 Reference Variables and Reference Types

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.

object vs. object reference variable



#### 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 longer-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.

array object



#### 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.

### 9.5.2 Accessing an Object's Data and Methods

dot operator (.)

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*:

- `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 is given in Section 9.7, Static Variables, Constants, and Methods.

invoking methods



### Note

Usually you create an object and assign it to a variable, and 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

## 9.5.3 Reference Data Fields and the `null` Value

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.

reference data fields

```
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'
}
```

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 value

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:

default field values

```
class Test {
    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 Test {
    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



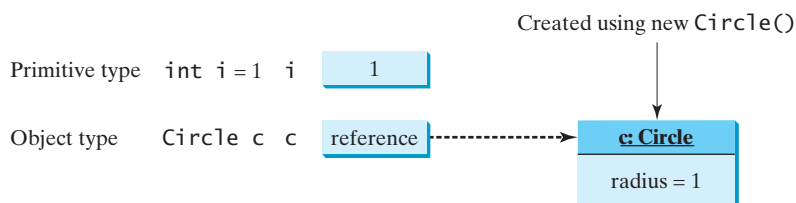
### Caution

**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 9.11c).

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

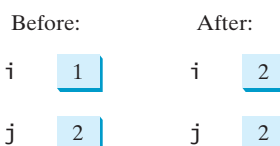
Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where 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.

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**



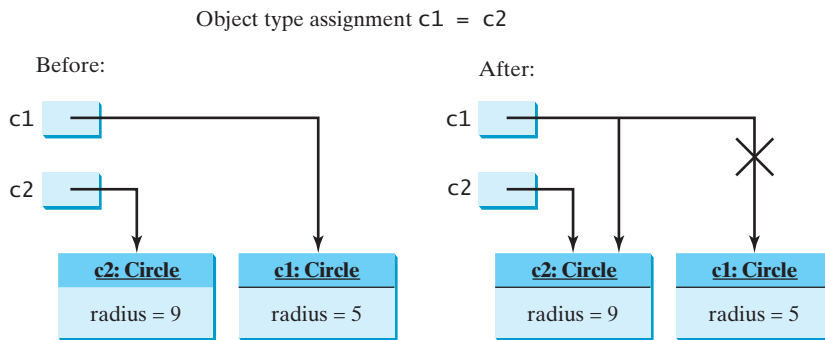
**FIGURE 9.7** A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

Primitive type assignment **i = j**



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

for 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.



- 9.7** Which operator is used to access a data field or invoke a method from an object?
- 9.8** What is an anonymous object?
- 9.9** What is `NullPointerException`?
- 9.10** 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.
- 9.11** 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         ShowErrors t = new ShowErrors();
4         t.x();
5     }
6 }
```

(b)

```
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 }
```

(c)

```
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 }
```

(d)

**9.12** 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 }

```

**9.13** 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



**Key  
Point**

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

Listing 9.1 defined the `SimpleCircle` 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.



**VideoNote**  
Use classes

### 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

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

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