The (non-`static`) methods of a class are called the **instance methods** of the class.

NOTE

In the definition of the `class` `Clock`, all the data members are `private` and all the method members are `public`. However, a method can also be `private`. For example, if a method is only used to support other methods of the class, and the user of the class does not need to access this method, you make it `private`.

Notice that we have not yet written the definitions of the methods of the `class` `Clock`. (You will learn how to write them in the section Definitions of the Constructors and Methods of the `class` `Clock`.) Also notice that the method `equals` has only one parameter, although you need two things to make a comparison. Similarly, the method `makeCopy` has only one parameter. An example later in this chapter will help explain why.

Before giving the definition of the `class` `Clock`, we first introduce another important concept related to classes—constructors.

Constructors

In addition to the methods necessary to implement operations, every class can have *special* types of methods called constructors. A **constructor** has the same name as the class, and it executes automatically when an object of that class is created. Constructors are used to guarantee that the instance variables of the class are initialized.

There are two types of constructors: those with parameters and those without parameters. The constructor without parameters is called the **default constructor**.

Constructors have the following properties:

- The name of a constructor is the same as the name of the class.
- A constructor, even though it is a method, has no return type. That is, it is neither a value-returning method nor a `void` method.
- A class can have more than one constructor. However, all constructors of a class have the same name. That is, the constructors of a class can be overloaded.
- If a class has more than one constructor, the constructors must have different *signatures*.
- Constructors execute automatically when class objects are instantiated. Because they have no types, they cannot be called like other methods.
- If there are multiple constructors, the constructor that executes depends on the type of values passed to the class object when the class object is instantiated.

For the `class` `Clock`, we will include two constructors: the default constructor and a constructor with parameters. The default constructor initializes the instance variables used to store the hours, minutes, and seconds, each to `0`. Similarly, the constructor with parameters initializes the instance variables to the values specified by the user. We will illustrate shortly how constructors are invoked.

The heading of the default constructor is:

```
public Clock()
```

The heading of the constructor with parameters is:

```
public Clock(int hours, int minutes, int seconds)
```

The definition of the `class` `Clock` has 16 members: 11 methods to implement the 11 operations, 2 constructors, and 3 instance variables to store the hours, minutes, and seconds.

NOTE

If you do not include any constructor in a class, then Java *automatically* provides the default constructor. Therefore, when you create an object, the instance variables are initialized to their default values. For example, `int` variables are initialized to `0`. If you provide at least one constructor and do not include the default constructor, then Java *will not automatically* provide the default constructor. Generally, if a class includes constructors, you should also include the default constructor.

Unified Modeling Language Class Diagrams

A class and its members can be described graphically using **Unified Modeling Language (UML)** notation. For example, Figure 8-1 shows the UML diagram of the `class` `Clock`. Also, what appears in the figure is called the **UML class diagram** of the class.

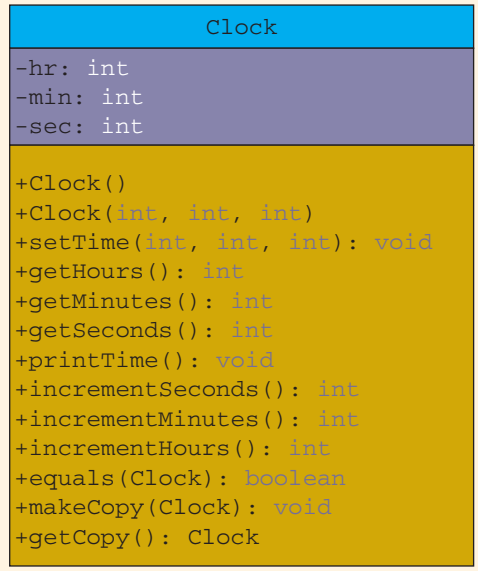

FIGURE 8-1 UML class diagram of the `class` `clock`

The top box in the UML diagram contains the name of the class. The middle box contains the data members and their data types. The last box contains the method names, parameter list, and return types. The + (plus) sign in front of a member indicates that it is a `public` member; the – (minus) sign indicates that it is a `private` member. The # symbol before a member name indicates that it is a `protected` member.

Variable Declaration and Object Instantiation

Once a `class` is defined, you can declare reference variables of that `class` type. For example, the following statements declare `myClock` and `yourClock` to be reference variables of type `Clock`:

```
Clock myClock;          //Line 1
Clock yourClock;        //Line 2
```

These statements *do not* allocate memory spaces to store the hours, minutes, and seconds. Next, we explain how to allocate memory space to store the hours, minutes, and seconds, and how to access that memory space using the variables `myClock` and `yourClock`.

The `class Clock` has three instance variables. To store the hours, minutes, and seconds, we need to create a `Clock` object, which is accomplished by using the operator `new`.

The general syntax for using the operator `new` is:

```
new className()                                          //Line 3
```

or:

```
new className(argument1, argument2, ..., argumentN)     //Line 4
```

The expression in Line 3 instantiates the object and initializes the instance variables of the object using the default constructor. The expression in Line 4 instantiates the object and initializes the instance variables using a constructor with parameters.

For the expression in Line 4:

- The number of arguments and their type should match the formal parameters (in the order given) of one of the constructors.
- If the type of the arguments does not match the formal parameters of any constructor (in the order given), Java uses type conversion and looks for the best match. For example, an integer value might be converted to a floating-point value with a zero decimal part. Any ambiguity will result in a compile-time error.

Consider the following statements (notice that `myClock` and `yourClock` are as declared in Lines 1 and 2):

```
myClock = new Clock();                          //Line 5
yourClock = new Clock(9, 35, 15);               //Line 6
```

The statement in Line 5 allocates memory space for a `Clock` object, initializes each instance variable of the object to `0`, and stores the address of the object into `myClock`. The statement in Line 6 allocates memory space for a `Clock` object; initializes the instance variables `hr`, `min`, and `sec` of the object to `9`, `35`, and `15`, respectively; and stores the address of the object into `yourClock` (see Figure 8-2).

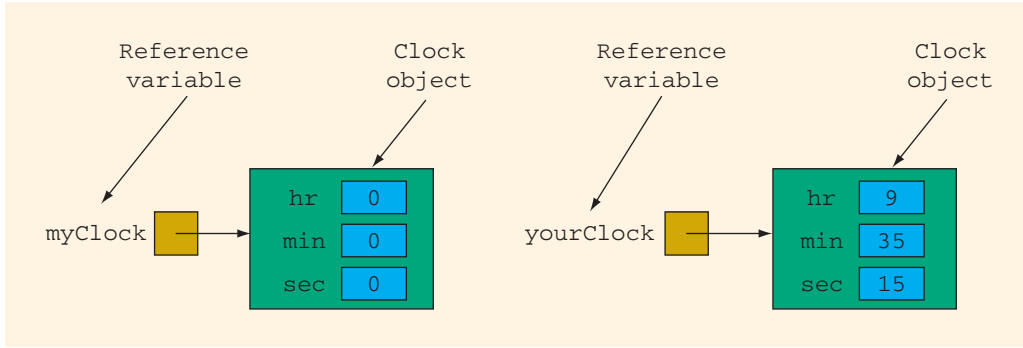

FIGURE 8-2 Variables `myClock` and `yourClock` and associated `Clock` objects

To be specific, we call the object to which `myClock` points the object `myClock` and the object to which `yourClock` points the object `yourClock` (see Figure 8-3).

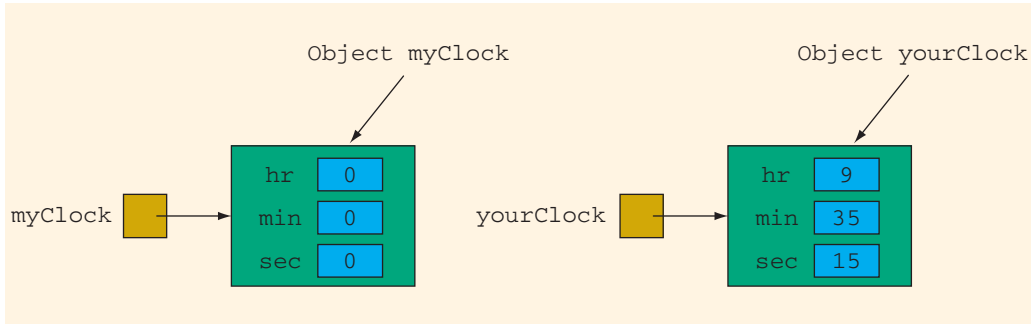

FIGURE 8-3 Objects `myClock` and `yourClock`

Of course, you can combine the statements to declare the variable and instantiate the object into one statement. For example, the statements in Lines 1 and 5 can be combined as:

```
Clock myClock = new Clock();    //declare and instantiate myClock
```

That is, the preceding statement declares `myClock` to be a reference variable of type `Clock` and instantiates the object `myClock` to store the hours, minutes, and seconds. Each instance variable of the object `myClock` is initialized to `0` by the default constructor.

Similarly, the statements in Lines 2 and 6 can be combined as:

```
Clock yourClock = new Clock(9, 35, 15);  //declare and
                                         //instantiate yourClock
```

That is, the preceding statement declares `yourClock` to be a reference variable of type `Clock` and instantiates the object `yourClock` to store the hours, minutes, and seconds. The instance variables `hr`, `min`, and `sec` of the object `yourClock` are initialized to `9`, `35`, and `15`, respectively, by the constructor with parameters.

NOTE

When we use phrases such as "create an object of a `class` type" we mean to: (i) declare a reference variable of the `class` type, (ii) instantiate the `class` object, and (iii) store the address of the object into the reference variable declared. For example, the following statements create the object `tempClock` of the `Clock` type:

```
Clock tempClock = new Clock();
```

The object `tempClock` is accessed via the reference variable `tempClock`.

Recall from Chapter 3 that a `class` object is called an **instance** of that `class`.

Accessing Class Members

Once an object of a class is created, the object can access the members (as explained in the next paragraph, after the syntax) of the `class`. The general syntax for an object to access a data member or a method is:

```
referenceVariableName.memberName
```

The class members that the class object can access depend on where the object is created.

- If the object is created in the definition of a method of the class, then the object can access both the `public` and `private` members. We will elaborate on this when we write the definitions of the methods `equals`, `makeCopy`, and `getCopy` of the `class` `Clock` later in this chapter.
- If the object is created elsewhere (for example, in a user's program), then the object can access *only* the `public` members of the class.

Recall that in Java, the dot `.` (period) is called the **member access operator**.

Example 8-1 illustrates how to access the members of a class.

EXAMPLE 8-1

Suppose that the objects `myClock` and `yourClock` have been created as before. Consider the following statements:

```
myClock.setTime(5, 2, 30);
myClock.printTime();
yourClock.setTime(x, y, z);  //Assume x, y, and z are variables
                             //of type int that have been
                             //initialized.
```

```
if (myClock.equals(yourClock))
.
.
.
```

These statements are legal; that is, they are syntactically correct. Note the following:

- In the first statement, `myClock.setTime(5, 2, 30);`, the method `setTime` is executed. The values `5`, `2`, and `30` are passed as parameters to the method `setTime`, and the method uses these values to set the values of `hr`, `min`, and `sec` of the object `myClock` to `5`, `2`, and `30`, respectively.
- Similarly, the second statement executes the method `printTime` and outputs the values of `hr`, `min`, and `sec` of the object `myClock`.
- In the third statement, the values of the variables `x`, `y`, and `z` are used to set the values of `hr`, `min`, and `sec` of the object `yourClock`.
- In the fourth statement, the method `equals` executes and compares the instance variables of the object `myClock` with the corresponding instance variables of the object `yourClock`. Because in this statement the method `equals` is invoked by the variable `myClock`, it has direct access to the instance variables of the object `myClock`. So it needs one more object to compare, which, in this case, is the object `yourClock`. This explains why the method `equals` has only one parameter.

The objects `myClock` and `yourClock` can access only `public` members of the class. The following statements are illegal because `hr` and `min` are `private` members of the `class` `Clock` and, therefore, cannot be accessed by `myClock` and `yourClock`:

```
myClock.hr = 10;                  //illegal
myClock.min = yourClock.min;      //illegal
```

Built-in Operations on Classes

Most of Java's built-in operations do not apply to classes. You cannot perform arithmetic operations on class objects. For example, you cannot use the operator `+` to add the values of two `Clock` objects. Also, you cannot use relational operators to compare two class objects in any meaningful way.

The built-in operation that is valid for classes is the dot operator (`.`). A reference variable uses the dot operator to access `public` members; classes can use the dot operator to access `public static` members.

Assignment Operator and Classes: A Precaution

This section discusses how the assignment operator works with reference variables and objects.

Suppose that the objects `myClock` and `yourClock` are as shown in Figure 8-4.

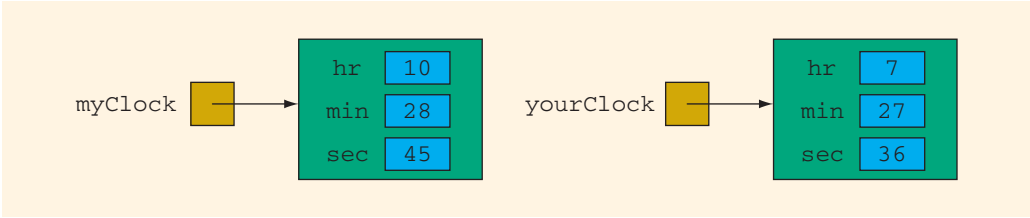

FIGURE 8-4 `myClock` and `yourClock`

The statement:

```
myClock = yourClock;
```

copies the value of the reference variable `yourClock` into the reference variable `myClock`. After this statement executes, both `yourClock` and `myClock` refer to the same object. Figure 8-5 illustrates this situation.

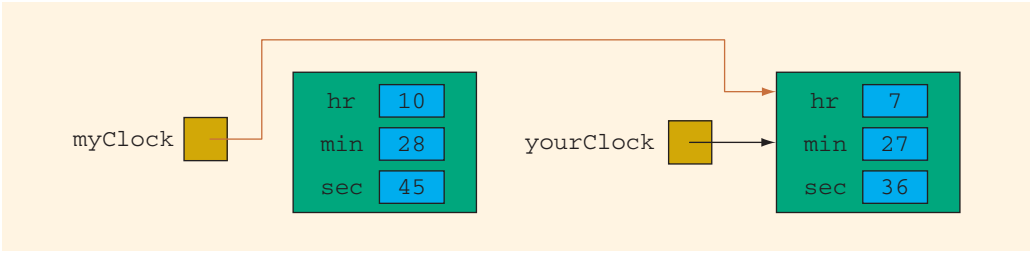

FIGURE 8-5 `myClock` and `yourClock` after the statement `myClock = yourClock;` executes

This is called the shallow copying of data. In **shallow copying**, two or more reference variables of the same type point to the same object; that is, two or more reference variables become aliases. Note that the object originally referred to by `myClock` becomes inaccessible.

To copy the instance variables of the object `yourClock` into the corresponding instance variables of the object `myClock`, you need to use the method `makeCopy`. This is accomplished by the following statement:

```
myClock.makeCopy(yourClock);
```

After this statement executes:

1. The value of `yourClock.hr` is copied into `myClock.hr`.
2. The value of `yourClock.min` is copied into `myClock.min`.
3. The value of `yourClock.sec` is copied into `myClock.sec`.

In other words, the values of the three instance variables of the object `yourClock` are copied into the corresponding instance variables of the object `myClock`, as shown in Figure 8-6.

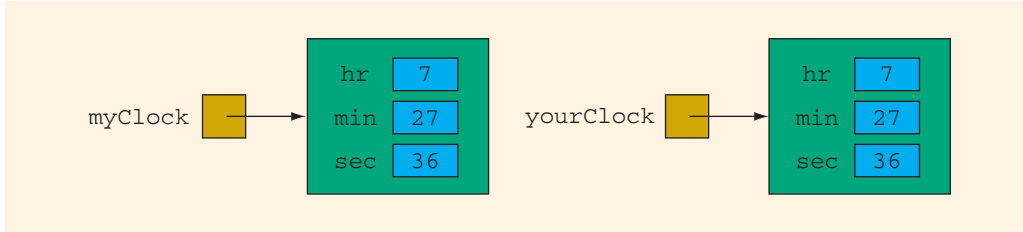

FIGURE 8-6 Objects `myClock` and `yourClock` after the statement `myClock.makeCopy(yourClock);` executes

This is called the deep copying of data. In **deep copying**, each reference variable refers to its *own* object, as in Figure 8-6, *not* the same object, as in Figure 8-5.

Another way to avoid the shallow copying of data is to have the object being copied create a copy of itself, and then return a reference to the copy. This is accomplished by the method `getCopy`. Consider the following statement:

```
myClock = yourClock.getCopy();
```

In this statement, the expression `yourClock.getCopy()` makes a copy of the object `yourClock` and returns the address, that is, the reference, of the copy. The assignment statement stores this address into `myClock`.

NOTE

The methods `makeCopy` and `getCopy` are both used to avoid the shallow copying of data. The main difference between these two methods is: To use the method `makeCopy`, both objects—the object whose data is being copied and the object that is copying the data—must be instantiated before invoking this method. To use the method `getCopy`, the object whose data is being copied must be instantiated before invoking this method, while the object of the reference variable receiving a copy of the data need not be instantiated. Note that `makeCopy` and `getCopy` are *user-defined* methods.

It is important to understand the difference between the shallow and deep copying of data and when to use which. Shallow copying can produce unintended results, especially by beginning Java programmers.

Class Scope

A reference variable follows the same scope rules as other variables. A member of a class is local to the class. You access a `public class` member outside the `class` through the reference variable name or the `class` name (for `static` members) and the member access operator (`.`).

Methods and Classes

Reference variables can be passed as parameters to methods and returned as method values. Recall from Chapter 7 that when a reference variable is passed as a parameter to a method, both the formal and actual parameters point to the same object.

Definitions of the Constructors and Methods of the `class` `Clock`

We now give the definitions of the methods of the `class` `Clock`, then we will write the complete definition of this class. First, note the following:

1. The `class` `Clock` has 11 methods: `setTime`, `getHours`, `getMinutes`, `getSeconds`, `printTime`, `incrementHours`, `incrementMinutes`, `incrementSeconds`, `equals`, `makeCopy`, and `getCopy`. It has two constructors and three instance variables: `hr`, `min`, and `sec`.
2. The three instance variables—`hr`, `min`, and `sec`—are `private` to the `class` and cannot be accessed directly outside the `class`.
3. The 11 methods—`setTime`, `getHours`, `getMinutes`, `getSeconds`, `printTime`, `incrementHours`, `incrementMinutes`, `incrementSeconds`, `equals`, `makeCopy`, and `getCopy`—can directly access the instance variables (`hr`, `min`, and `sec`). In other words, we do not pass instance variables or data members as parameters to these methods. Similarly, constructors directly access the instance variables.

Let's first write the definition of the method `setTime`. The method `setTime` has three parameters of type `int`. This method sets the instance variables to the values specified by the user, which are passed as parameters to this function. The definition of the method `setTime` follows:

```
public void setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

Note that the definition of the method `setTime` checks for the valid values of `hours`, `minutes`, and `seconds`. If any of these values is out of range, the corresponding instance variable is initialized to `0`. Now let's look at how the method `setTime` works.

The method `setTime` is a `void` method and has three parameters. Therefore:

- A call to this method is a stand-alone statement.
- We must use three parameters in a call to this method.

Furthermore, recall that because `setTime` is a member of the `class` `Clock`, it can directly access the instance variables `hr`, `min`, and `sec`, as shown in the definition of `setTime`.

Suppose that the object `myClock` is as shown in Figure 8-7.

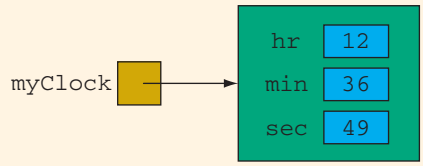

FIGURE 8-7 Object `myClock`

Consider the following statement:

```
myClock.setTime(3, 48, 52);
```

The variable `myClock` accesses the member `setTime`. In the statement `myClock.setTime(3, 48, 52);`, `setTime` is accessed by the variable `myClock`. Therefore, the three variables—`hr`, `min`, and `sec`—referred to in the body of the method `setTime` are the three instance variables of the object `myClock`. Thus, the values `3`, `48`, and `52`, which are passed as parameters in the preceding statement, are assigned to the three instance variables of the object `myClock` by the method `setTime` (see the body of the method `setTime`). After the previous statement executes, `myClock` is as shown in Figure 8-8.

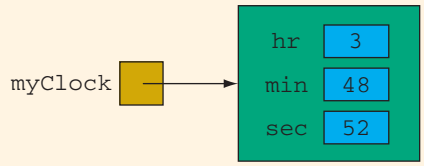

FIGURE 8-8 `myClock` after statement `myClock.setTime(3,48,52);` executes

Next, let's give the definitions of the other methods of the class Clock. These definitions are simple and easy to follow.

```
public int getHours()
{
    return hr;          //return the value of hr
}

public int getMinutes()
{
    return min;         //return the value of min
}

public int getSeconds()
{
    return sec;         //return the value of sec
}

public void printTime()
{
    if (hr < 10)
       System.out.print("0");
    System.out.print(hr + ":");

    if (min < 10)
       System.out.print("0");
    System.out.print(min + ":");

    if (sec < 10)
       System.out.print("0");
    System.out.print(sec);
}

public void incrementHours()
{
    hr++;            //increment the value of hr by 1

    if (hr > 23)     //if hr is greater than 23,
        hr = 0;      //set hr to 0
}

public void incrementMinutes()
{
    min++;            //increment the value of min by 1

    if (min > 59)      //if min is greater than 59
    {
        min = 0;            //set min to 0
        incrementHours();  //increment hours
    }
}
```

```
public void incrementSeconds()
{
    sec++;            //increment the value of sec by 1

    if (sec > 59)     //if sec is greater than 59
    {
        sec = 0;              //set sec to 0
        incrementMinutes();   //increment minutes
    }
}
```

From the definitions of the methods `incrementMinutes` and `incrementSeconds`, you can see that a method of a `class` can call other methods of the `class`.

The method `equals` has the following definition:

```
public boolean equals(Clock otherClock)
{
    return (hr == otherClock.hr
            && min == otherClock.min
            && sec == otherClock.sec);
}
```

Let's see how the method `equals` works.

Suppose that `myClock` and `yourClock` are as shown in Figure 8-9.

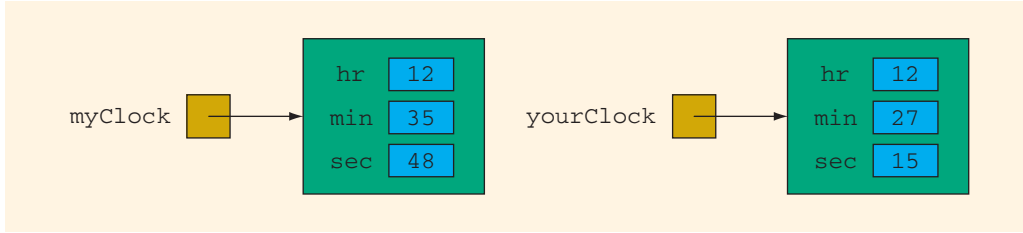

FIGURE 8-9 Objects `myClock` and `yourClock`

Consider the following statement:

```
if (myClock.equals(yourClock))
.
.
.
```

In the expression:

```
myClock.equals(yourClock)
```

`myClock` accesses the method `equals`. The value of the parameter `yourClock` is passed to the formal parameter `otherClock`, as shown in Figure 8-10.

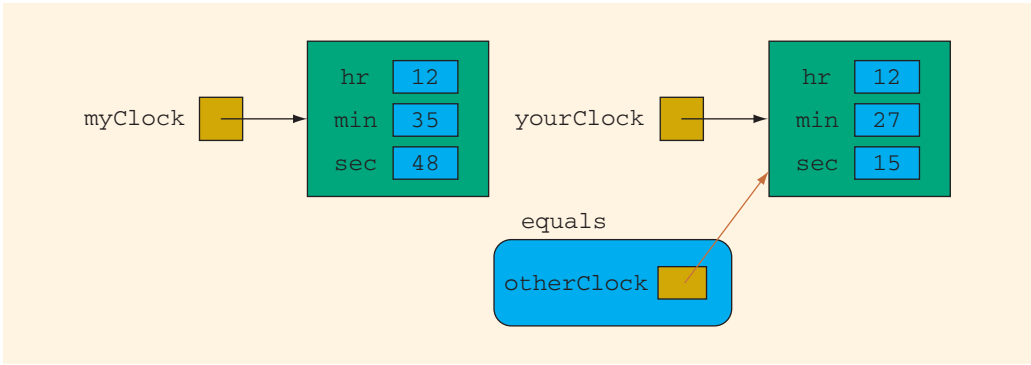

FIGURE 8-10 Object `myClock` and parameter `otherClock`

Note that `otherClock` and `yourClock` refer to the same object. The instance variables `hr`, `min`, and `sec` of the object `otherClock` have the values `12`, `27`, and `15`, respectively. In other words, when the body of the method `equals` executes, the value of `otherClock.hr` is `12`, the value of `otherClock.min` is `27`, and the value of `otherClock.sec` is `15`. The method `equals` is a member of `myClock`. When the method `equals` executes, the variables `hr`, `min`, and `sec` in the body of the method `equals` are the instance variables of the object `myClock`. Therefore, the instance variable `hr` of the object `myClock` is compared with `otherClock.hr`, the instance variable `min` of the object `myClock` is compared with `otherClock.min`, and the instance variable `sec` of the object `myClock` is compared with `otherClock.sec`.

Once again, in the expression:

```
myClock.equals(yourClock)
```

the method `equals` is invoked by `myClock` and compares the object `myClock` with the object `yourClock`. It follows that the method `equals` needs only one parameter.

Let us again take a look at the definition of the method `equals`. Notice that within the definition of this method, the object `otherClock` accesses the data members `hr`, `min`, and `sec`. However, these data members are `private`. So is there any violation? The answer is no. The method `equals` is a member of the `class` `Clock` and `hr`, `min`, and `sec` are the data members. Moreover, `otherClock` is an object of the `class` `Clock`. Therefore, the object `otherClock` can access its `private` data members within the definition of the method `equals`. The same is true for any method of a class.

That is, in general, when you write the definition of a method, say, `dummyMethod`, of a `class`, say, `DummyClass`, and the method uses an object, `dummyObject` of the `class` `DummyClass`, then within the definition of `dummyMethod` the object `dummyObject` can access its `private` data members (in fact, any `private` member of the class).

The method `makeCopy` copies the instance variables of its parameter, `otherClock`, into the corresponding instance variables of the object referenced by the variable using this method. Its definition is:

```
public void makeCopy(Clock otherClock)
{
    hr = otherClock.hr;
    min = otherClock.min;
    sec = otherClock.sec;
}
```

Consider the following statement:

```
myClock.makeCopy(yourClock);
```

In this statement, the method `makeCopy` is invoked by `myClock`. The three instance variables `hr`, `min`, and `sec` in the body of the method `makeCopy` are the instance variables of the object `myClock`. The variable `yourClock` is passed as a parameter to `makeCopy`. Therefore, `yourClock` and `otherClock` refer to the same object, which is the object `yourClock`. Thus, after the preceding statement executes, the instance variables of the object `yourClock` are copied into the corresponding instance variables of the object `myClock`. (Note that as in the case of the method `equals`, the parameter `otherClock` can directly access the private data members of the object it points to.)

The method `getCopy` creates a copy of an object's `hr`, `min`, and `sec` and returns the address of the copy of the object. That is, the method `getCopy` creates a new `Clock` object, initializes the instance variables of the object, and returns the address of the object created. The definition of the method `getCopy` is:

```
public Clock getCopy()
{
    Clock temp = new Clock();    //Line 1

    temp.hr = hr;                //Line 2
    temp.min = min;              //Line 3
    temp.sec = sec;              //Line 4

    return temp;                 //Line 5
}
```

The following illustrates how the method `getCopy` works. Suppose that `yourClock` is as shown in Figure 8-11.

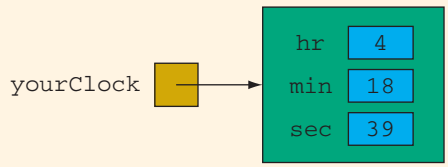

FIGURE 8-11 Object `yourClock`

Consider the following statement:

```
myClock = yourClock.getCopy();          //Line A
```

In this statement, because the method `getCopy` is invoked by `yourClock`, the three variables `hr`, `min`, and `sec` in the body of the method `getCopy` are the instance variables of the object `yourClock`. The body of the method `getCopy` executes as follows. The statement in Line 1 creates the `Clock` object `temp`. The statements in Lines 2 through 4 copy the instance variables of the object `yourClock` into the corresponding instance variables of `temp`. In other words, the object referenced by `temp` is a copy of the object `yourClock` (see Figure 8-12).

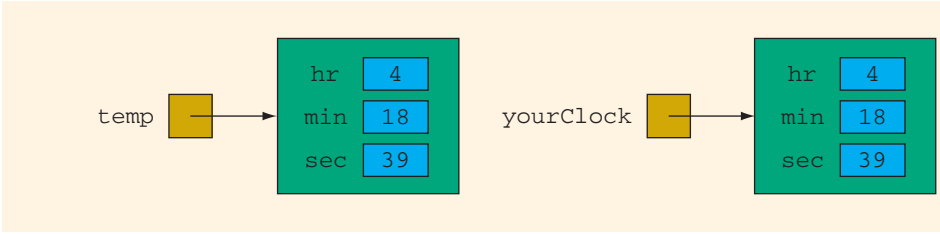

FIGURE 8-12 Objects `temp` and `yourClock`

The statement in Line 5 returns the value of `temp`, which is the address of the object holding a copy of the data. The value returned by the method `getCopy` is copied into `myClock`. Therefore, after the statement in Line A executes, `myClock` and `yourClock` are as shown in Figure 8-13.

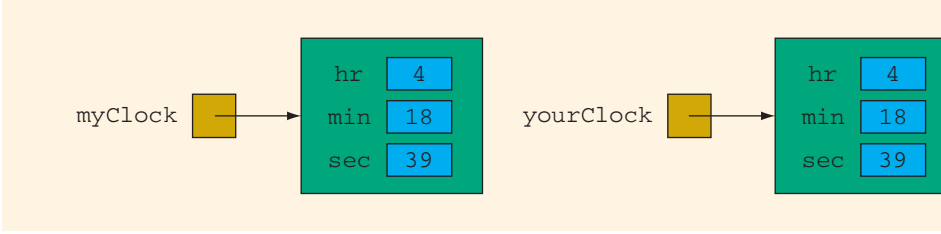

FIGURE 8-13 Objects `myClock` and `yourClock`

Note that as in the case of the methods `equals` and `makeCopy`, the reference variable `temp`—in the definition of the method `getCopy`—can directly access the private data members of the object it points to because `getCopy` is a method of the `class Clock`.

NOTE

The definition of the method `getCopy` can also be written as:

```
public Clock getCopy()
{
    Clock temp = new Clock(hr, min, sec);

    return temp;
}
```

This definition of the method `getCopy` uses the constructor with parameters, described below, to initialize the instance variables of the object `temp`.

Next, we give the definitions of the constructors. The default constructor initializes each instance variable to `0`. Its definition is:

```
public Clock()
{
    hr = 0;
    min = 0;
    sec = 0;
}
```

You can also write the definition of the default constructor using the method `setTime` as follows:

```
public Clock()
{
    setTime(0, 0, 0);
}
```

The definition of the constructor with parameters is the same as the definition of the method `setTime`. It initializes the instance variables to the values specified by the user. Its definition is:

```
public Clock(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

As in the case of the default constructor, you can write the definition of the constructor with parameters using the method `setTime` as follows:

```
public Clock(int hours, int minutes, int seconds)
{
    setTime(hours, minutes, seconds);
}
```

This definition of the constructor with parameters makes debugging easier, because only the code for the method `setTime` needs to be checked.

DEFINITION OF THE `Class Clock`

Now that we have defined the methods of the `class Clock`, we can give the complete definition of the `class Clock`. Before the definition of a method, we include comments specifying the preconditions and/or postconditions.

Precondition: A statement specifying the condition(s) that must be true before the function is called.

Postcondition: A statement specifying what is true after the function call is completed.

The definition of the `class Clock` is:

```
public class Clock
{
    private int hr;  //store hours
    private int min; //store minutes
    private int sec; //store seconds

        //Default constructor
        //Postcondition: hr = 0; min = 0; sec = 0
    public Clock()
    {
        setTime(0, 0, 0);
    }

        //Constructor with parameters, to set the time
        //The time is set according to the parameters.
        //Postcondition: hr = hours; min = minutes;
        //               sec = seconds
    public Clock(int hours, int minutes, int seconds)
    {
        setTime(hours, minutes, seconds);
    }


        //Method to set the time
        //The time is set according to the parameters.
        //Postcondition: hr = hours; min = minutes;
        //               sec = seconds
```

```
public void setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}

    //Method to return the hours
    //Postcondition: the value of hr is returned
public int getHours()
{
    return hr;
}

    //Method to return the minutes
    //Postcondition: the value of min is returned
public int getMinutes()
{
    return min;
}

    //Method to return the seconds
    //Postcondition: the value of sec is returned
public int getSeconds()
{
    return sec;
}

    //Method to print the time
    //Postcondition: Time is printed in the form hh:mm:ss
public void printTime()
{
    if (hr < 10)
        System.out.print("0");
    System.out.print(hr + ":");

    if (min < 10)
        System.out.print("0");
    System.out.print(min + ":");
```

```
    if (sec < 10)
        System.out.print("0");
    System.out.print(sec);
}

    //Method to increment the time by one second
    //Postcondition: The time is incremented by one second
    //If the before-increment time is 23:59:59, the time
    //is reset to 00:00:00
public void incrementSeconds()
{
    sec++;

    if (sec > 59)
    {
        sec = 0;
        incrementMinutes(); //increment minutes
    }
}

    //Method to increment the time by one minute
    //Postcondition: The time is incremented by one minute
    //If the before-increment time is 23:59:53, the time
    //is reset to 00:00:53
public void incrementMinutes()
{
    min++;

    if (min > 59)
    {
        min = 0;
        incrementHours(); //increment hours
    }
}

    //Method to increment the time by one hour
    //Postcondition: The time is incremented by one hour
    //If the before-increment time is 23:45:53, the time
    //is reset to 00:45:53
public void incrementHours()
{
    hr++;

    if (hr > 23)
        hr = 0;
}

    //Method to compare two times
    //Postcondition: Returns true if this time is equal to
    //               otherClock; otherwise returns false
public boolean equals(Clock otherClock)
```

```
    {
         return (hr == otherClock.hr
                && min == otherClock.min
                && sec == otherClock.sec);
    }

        //Method to copy time
        //Postcondition: The instance variables of otherClock
        //               copied into the corresponding data
        //               are members of this time.
        //               hr = otherClock.hr;
        //               min = otherClock.min;
        //               sec = otherClock.sec;
    public void makeCopy(Clock otherClock)
    {
        hr = otherClock.hr;
        min = otherClock.min;
        sec = otherClock.sec;
    }

        //Method to return a copy of time
        //Postcondition: A copy of the object is created and
        //               a reference of the copy is returned
    public Clock getCopy()
    {
        Clock temp = new Clock();

        temp.hr = hr;
        temp.min = min;
        temp.sec = sec;

        return temp;
    }
}
```

NOTE In a class definition, it is a common practice to list all the instance variables, named constants, other data members, or variable declarations first, then the constructors, and then the methods.

Once a class is properly defined and implemented, it can be used in a program. A program or software that uses and manipulates the objects of a class is called a **client** of that class.

EXAMPLE 8-2

```
//Program to test various operations of the class Clock

import java.util.*;

public class TestProgClock
```

```
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        Clock myClock = new Clock(5, 4, 30);          //Line 1
        Clock yourClock = new Clock();                //Line 2

        int hours;                                    //Line 3
        int minutes;                                  //Line 4
        int seconds;                                  //Line 5

        System.out.print("Line 6: myClock: ");        //Line 6
        myClock.printTime();                          //Line 7
        System.out.println();                         //Line 8
        System.out.print("Line 9: yourClock: ");      //Line 9
        yourClock.printTime();                        //Line 10
        System.out.println();                         //Line 11

        yourClock.setTime(5, 45, 16);                 //Line 12

        System.out.print("Line 13: After setting "
                       + "the time - yourClock: ");   //Line 13
        yourClock.printTime();                        //Line 14
        System.out.println();                         //Line 15

        if (myClock.equals(yourClock))                //Line 16
            System.out.println("Line 17: Both the "
                             + "times are equal.");   //Line 17
        else                                          //Line 18
            System.out.println("Line 19: The two "
                             + "times are not "
                             + "equal.");             //Line 19

        System.out.print("Line 20: Enter hours, "
                       + "minutes, and seconds: ");   //Line 20
        hours = console.nextInt();                    //Line 21
        minutes = console.nextInt();                  //Line 22
        seconds = console.nextInt();                  //Line 23
        System.out.println();                         //Line 24

        myClock.setTime(hours, minutes, seconds);     //Line 25

        System.out.print("Line 26: New time of "
                       + "myClock: ");                //Line 26
        myClock.printTime();                          //Line 27
        System.out.println();                         //Line 28

        myClock.incrementSeconds();                   //Line 29

        System.out.print("Line 30: After "
                       + "incrementing the time by "
                       + "one second, myClock: ");    //Line 30
```

```
        myClock.printTime();                                    //Line 31
        System.out.println();                                   //Line 32

        yourClock.makeCopy(myClock);                            //Line 33

        System.out.print("Line 34: After copying "
                       + "myClock into yourClock, "
                       + "yourClock: ");                        //Line 34
        yourClock.printTime();                                  //Line 35
        System.out.println();                                   //Line 36
    }//end main
}
```

Sample Run: (In this sample run, the user input is shaded.)

```
Line 6: myClock: 05:04:30
Line 9: yourClock: 00:00:00
Line 13: After setting the time - yourClock: 05:45:16
Line 19: The two times are not equal.
Line 20: Enter hours, minutes, and seconds: 11 22 59

Line 26: New time of myClock: 11:22:59
Line 30: After incrementing the time by one second, myClock: 11:23:00
Line 34: After copying myClock into yourClock, yourClock: 11:23:00
```

A walk-through of the preceding program is left as an exercise for you.

Classes and the Method `toString`

Suppose that `x` is an `int` variable and the value of `x` is `25`. The statement:

```
System.out.println(x);
```

outputs:

```
25
```

However, the output of the statement:

```
System.out.println(myClock);
```

is:

```
Clock@11b86e7
```

which looks strange. (Note that when you execute a similar statement, you are likely to get a different but similar output.) This is because whenever you create a `class`, the Java system provides the method `toString` to that `class`. The method `toString` is used to convert an object to a `String` object. When an object reference is provided as a parameter to the methods `print`, `println`, and `printf`, the `toString` method is called.

The default definition of the method toString creates a string that is the name of the object's class, followed by the hash code of the object. For example, in the preceding statement, Clock is the name of the object myClock's class and the hash code for the object referenced by myClock is @11b86e7.

The method toString is a public value-returning method. It does not take any parameters and returns the address of a String object. The heading of the method toString is:

```
public String toString()
```

You can *override* the default definition of the method toString to convert an object to a desired string. Suppose that for the objects of the class Clock you want the method toString to create the string hh:mm:ss—the string consists of the object's hour, minutes, seconds, and the colons as shown. The string created by the method toString is the same as the string output by the method printTime of the class Clock. This is easily accomplished by providing the following definition of the method toString:

```
public String toString()
{
    String str = "";

    if (hr < 10)
        str = "0";
    str = str + hr + ":";

    if (min < 10)
        str = str + "0" ;
    str = str + min + ":";

    if (sec < 10)
        str = str + "0";
    str = str + sec;

    return str;
}
```

In the preceding code, str is a String variable used to create the required string.

The preceding definition of the method toString must be included in the class Clock. In fact, after including the method toString in the class Clock, we can remove the method printTime. If the values of the instance variables hr, min, and sec of myClock are 8, 25, and 56, respectively, then the output of the statement:

```
System.out.println(myClock)
```

is:

```
08:25:56
```

You can see that the method toString is useful for outputting the values of the instance variables. Note that the method toString only returns the (formatted) string; the methods print, println, or printf output the string.

EXAMPLE 8-3

In this example, we give the complete definition of the `class` `Circle`, which was briefly discussed in the beginning of this chapter.

```
public class Circle
{
    private double radius;

      //Default constructor
      //Sets the radius to 0
    Circle()
    {
        radius = 0;
    }

      //Constructor with a parameter
      //Sets the radius to the value specified by the parameter r.
    Circle(double r)
    {
        radius = r;
    }

      //Method to set the radius of the circle.
      //Sets the radius to the value specified by the parameter r.
    public void setRadius(double r)
    {
        radius = r;
    }

      //Method to return the radius of the circle.
      //Returns the radius of the circle.
    public double getRadius()
    {
        return radius;
    }

      //Method to compute and return the area of the circle.
      //Computes and returns the area of the circle.
    public double area()
    {
       return Math.PI * Math.PI * radius;
    }

      //Method to compute and return the perimeter of the circle.
      //Computes and returns the area of the circle.
    public double perimeter()
    {
       return 2 * Math.PI * radius;
    }

      //Method to return the radius, area, perimeter of the circle
      //as a string.
```

```
    public String toString()
    {
        return String.format("Radius = %.2f, Perimeter = %.2f"
                    + ", Area = %.2f%n", radius, perimeter(),
                    area());
    }
}
```

We leave the UML class diagram of the class Circle as an exercise for you.

The following program shows how to use the class Circle in a program.

```
// Program to test various operations of the class Circle.

import java.util.*;                                          //Line 1

public class TestProgCircle                                  //Line 2
{                                                            //Line 3
    static Scanner console = new Scanner(System.in);         //Line 4

    public static void main(String[] args)                   //Line 5
    {                                                        //Line 6
        Circle firstCircle = new Circle();                   //Line 7
        Circle secondCircle = new Circle(12);                //Line 8

        double radius;                                       //Line 9

        System.out.println("Line 10: firstCircle: "
                         + firstCircle);                     //Line 10

        System.out.println("Line 11: secondCircle: "
                         + secondCircle);                    //Line 11

        System.out.print("Line 12: Enter the radius: ");     //Line 12
        radius = console.nextDouble();                       //Line 13
        System.out.println();                                //Line 14

        firstCircle.setRadius(radius);                       //Line 15

        System.out.println("Line 16: firstCircle: "
                        + firstCircle );                     //Line 16

        if (firstCircle.getRadius()
                  > secondCircle.getRadius())                //Line 17
            System.out.println("Line 18: The radius of "
                 + "the first circle is greater than "
                 + "the radius of the second circle. ");     //Line 18
        else if (firstCircle.getRadius()
                    < secondCircle.getRadius())              //Line 19
            System.out.println("Line 20: The radius of "
                 + "the first circle is less than the "
                 + "radius of the second circle. ");         //Line 20
```

8

```
        else                                                    //Line 21
            System.out.println("Line 22: The radius of "
                 + "both the circles are the same.");          //Line 22
    }//end main                                                 //Line 23
}                                                               //Line 24
```

Sample Run: (In this sample run, the user input is shaded.)

```
Line 10: firstCircle: Radius = 0.00, Perimeter = 0.00, Area = 0.00

Line 11: secondCircle: Radius = 12.00, Perimeter = 75.40, Area = 118.44

Line 12: Enter the radius: 10

Line 16: firstCircle: Radius = 10.00, Perimeter = 62.83, Area = 98.70

Line 20: The radius of the first circle is less than the radius of the
second circle.
```

The preceding program works as follows. The statement in Line 7 creates the object `firstCircle` and using the default constructor sets the radius to `0`. The statement in Line 8 creates the object `secondCircle` and sets the radius to `12`. The statement in Line 9 declares the `double` variable `radius`. The statement in Line 10 outputs the radius, area, and perimeter of the `firstCircle`. Similarly, the statement in Line 11 outputs the radius, area, and perimeter of the `secondCircle` The statement in Line 12 prompts the user to enter the value of `radius`. The statement in Line 13 stores the value entered by the user in the variable `radius`. The statement in Line 15 uses the value of `radius` to set the radius of `firstCircle`. The statement in Line 16 outputs the radius, area, and perimeter of the `firstCircle`. The statements in Lines 17 to 23 compare the radius of `firstCircle` and `secondCircle` and output the appropriate result.

EXAMPLE 8-4

In Example 7-3, the method `rollDice` rolls a pair of dice until the sum of the numbers rolled is a given number and returns the number of times the dice are rolled to get the desired sum. In fact, we can design a class that implements the basic properties of a die. Consider the definition of the following `class` `RollDie`.

```
public class RollDie
{
    private int num;

      //Default constructor
      //Sets the default number rolled by a die to 1
    RollDie()
    {
        num = 1;
    }
```

```
    //Method to roll a die.
    //This method uses a random number generator to randomly
    //generate a number between 1 and 6, and stores the number
    //in the instance variable num and returns the number.
  public int roll()
  {
      num = (int) (Math.random() * 6) + 1;

      return num;
  }

    //Method to return the number on the top face of the die.
    //Returns the value of the instance variable num.
  public int getNum()
  {
      return num;
  }

    //Returns the value of the instance variable num as a string.
  public String toString()
  {
      return "" + num;
  }
}
```

We leave the UML class diagram of the class RollDie as an exercise for you.

The following program shows how to use the class RollDie in a program.

```
// Program to test various operations of the class RollDie.

import java.util.*;                                          //Line 1

public class TestProgRollDie                                 //Line 2
{                                                            //Line 3
    static Scanner console = new Scanner(System.in);         //Line 4

    public static void main(String[] args)                   //Line 5
    {                                                        //Line 6
        RollDie die1 = new RollDie();                        //Line 7
        RollDie die2 = new RollDie();                        //Line 8

        System.out.println("Line 9: die1: " + die1);         //Line 9

        System.out.println("Line 10: die2: " + die2);        //Line 10

        System.out.println("Line 11: After rolling "
                         + "die1: " + die1.roll());          //Line 11

        System.out.println("Line 12: After rolling "
                         + "die2: " + die2.roll());          //Line 12

        System.out.println("Line 13: Sum of the "
                 + "numbers rolled by the dice is: "
                 + (die1.getNum() + die2.getNum()));         //Line 13
```

```
        System.out.println("Line 14: After again rolling "
                + "the sum of the numbers rolled is: "
                + (die1.roll() + die2.roll()));       //Line 14
    }//end main                                       //Line 15
}                                                     //Line 16
```

Sample Run:

```
Line 9: die1: 1
Line 10: die2: 1
Line 11: After rolling die1: 5
Line 12: After rolling die2: 3
Line 13: Sum of the numbers rolled by the dice is: 8
Line 14: After again rolling the sum of the numbers rolled is: 4
```

The preceding program works as follows. The statements in Lines 7 and 8 create the objects `die1` and `die2`, and using the default constructor set both the dice to 1. The statements in Lines 9 and 10 output the number of both the dice. The statement in Line 11 rolls `die1` and outputs the number rolled. Similarly, the statement in Line 12 rolls `die2` and outputs the number rolled. The statement in Line 13 outputs the sum of the numbers rolled by `die1` and `die2`. The statement in Line 14 again rolls both the dice and outputs the sum of the numbers rolled.

Copy Constructor

Suppose that you have the following statement:

```
Clock myClock = new Clock(8, 45, 22);         //Line 1
```

You can use the object `myClock` to declare and instantiate another `Clock` object. Consider the following statement:

```
Clock aClock = new Clock(myClock);            //Line 2
```

This statement declares `aClock` to be a reference variable of type `Clock`, instantiates the object `aClock`, and initializes the instance variables of the object `aClock` using the values of the corresponding instance variables of the object `myClock`. However, to successfully execute the statement in Line 2, you need to include a special constructor, called a **copy constructor**, in the `class Clock`. The copy constructor executes when an object is instantiated and initialized using an existing object.

The syntax of the heading of the copy constructor is:

```
public ClassName(ClassName otherObject)
```

For example, the heading of the copy constructor for the `class Clock` is:

```
public Clock(Clock otherClock)
```

The definition of the copy constructor for the class Clock is:

```
public Clock(Clock otherClock)
{
    hr = otherClock.hr;
    min = otherClock.min;
    sec = otherClock.sec;
}
```

If you include this definition of the copy constructor in the class Clock, then the statement in Line 2 declares aClock to be a reference variable of type Clock, instantiates the object aClock, and initializes the instance variables of the object aClock using the values of the instance variables of the object myClock.

NOTE

The definition of the copy constructor of the class Clock can also be written as:

```
public Clock(Clock otherClock)
{
    setTime(otherClock.hr, otherClock.min, otherClock.sec);
}
```

The copy constructor is useful and will be included in most of the classes.

8

Static Members of a Class

In Chapter 7, we described the classes Math and Character. In Example 7-1 (of Chapter 7), we used several methods of the classes Math and Character; however, we did not need to create any objects to use these methods. We simply used the import statement:

```
import static java.lang.Math.*;
```

and then called the method with an appropriate actual parameter list. For example, to use the method pow of the class Math, we used expressions such as:

```
pow(5, 3)
```

Recall from Chapter 7 that if you are using versions of Java lower than Java 5.0 or you do not include the preceding import statement, then you call the method pow as follows:

```
Math.pow(5, 3)
```

That is, we can simply call the method using the name of the class and the dot operator.

We cannot use the same approach with the class Clock. Although the methods of the class Math are public, they also are defined using the modifier static. For example, the heading of the method pow of the class Math is:

```
public static double pow(double base, double exponent)
```

The modifier static in the heading specifies that the method can be invoked by using the name of the class. Similarly, if a data member of a class is declared using the modifier static, it can be accessed by using the name of the class.

The following example clarifies the effect of the modifier `static`.

EXAMPLE 8-5

Consider the following definition of the `class` `Illustrate`:

```
public class Illustrate
{
    private int x;
    private static int y;
    public static int count;

       //Default constructor
       //Postcondition: x = 0;
    public Illustrate()
    {
        x = 0;
    }

       //Constructor with parameters
       //Postcondition: x = a;
    public Illustrate(int a)
    {
         x = a;
    }

        //Method to set x.
        //Postcondition: x = a;
    void setX(int a)
    {
        x = a;
    }

        //Method to return the values of the instance
        //and static variables as a string
        //The string returned is used by the methods
        //print, println, or printf to print the values
        //of the instance and static variables.
        //Postcondition: The values of x, y, and count
        //are returned as a string.
    public String toString()
    {
        return("x = " + x + ", y = " + y
             + ", count = " + count);
    }

       //Method to increment the value of the private
       //static member y
       //Postcondition: y is incremented by 1.
```

```
    public static void incrementY()
    {
        y++;
    }
}
```

Suppose that you have the following declaration:

```
Illustrate illusObject = new Illustrate();
```

The reference variable `illusObject` can access any `public` member of the `class` `Illustrate`.

The method `incrementY` is `static` and `public`, so the following statement is legal:

```
Illustrate.incrementY();
```

Similarly, because the data member `count` is `static` and `public`, the following statement is legal:

```
Illustrate.count++;
```

In essence, `public static` members of a `class` can be accessed either by an object, that is, by using a reference variable of the `class` type, or using the `class` name and the dot operator.

`static` Variables (Data Members) of a Class

Suppose that you have a `class`, say, `MyClass`, with data members (`static` and non-`static`). When you instantiate the objects of type `MyClass`, only the non-`static` data members of the `class` `MyClass` become the data members of each object. What about the memory for the `static` data members of `MyClass`? For each `static` data member of the `class`, Java allocates memory space only once. All `MyClass` objects refer to the same memory space. In fact, `static` data members of a `class` *exist* even when no object of the `class` type is instantiated. Moreover, `static` variables are initialized to their default values. You can access the `public static` data members outside the `class`, as explained in the previous section.

The following example further clarifies how memory space is allocated for `static` and non-`static` data members of a class.

Suppose that you have the `class` `Illustrate`, as given in Example 8-5. Then, memory space exists for the `static` data members `y` and `count`.

Consider the following statements:

```
Illustrate illusObject1 = new Illustrate(3);      //Line 1
Illustrate illusObject2 = new Illustrate(5);      //Line 2
```

The statements in Lines 1 and 2 declare `illusObject1` and `illusObject2` to be reference variables of type `Illustrate` and instantiate these objects (see Figure 8-14).

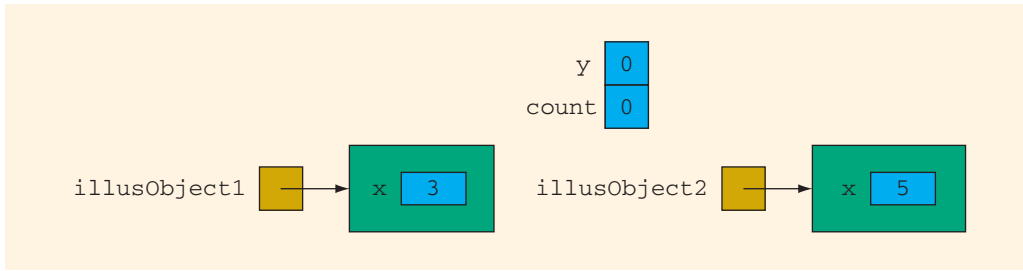

FIGURE 8-14 `illusObject1` and `illusObject2`

Now consider the following statement:

```
Illustrate.incrementY();
Illustrate.count++;
```

After these statements execute, the objects and static members are as shown in Figure 8-15.

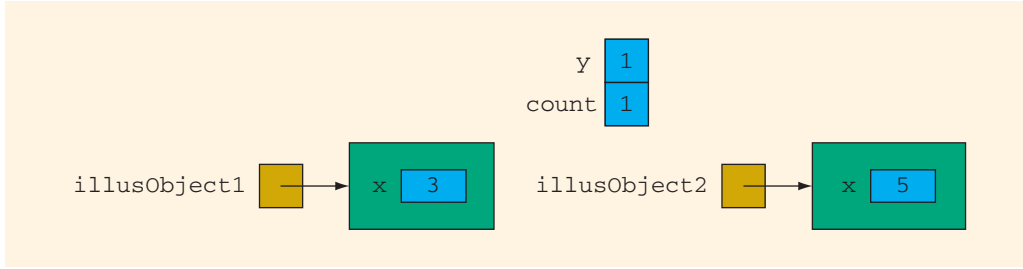

FIGURE 8-15 `illusObject1` and `illusObject2` after the statements `Illustrate.incrementY();` and `Illustrate.count++;` execute

The output of the statement:

```
System.out.println(illusObject1);          //Line 3
```

is:

```
x = 3, y = 1, count = 1
```

Similarly, the output of the statement:

```
System.out.println(illusObject2);          //Line 4
```

is:

```
x = 5, y = 1, count = 1
```

Now consider the statement:

```
Illustrate.count++;
```

After this statement executes, the objects and static members are as shown in Figure 8-16.

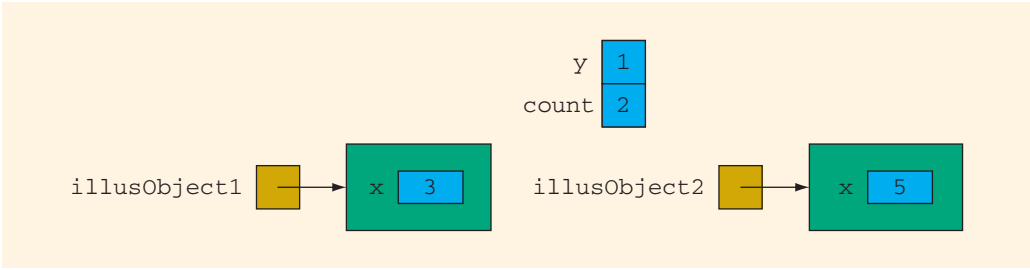

FIGURE 8-16 `illusObject1` and `illusObject2` after the statement `Illustrate.count++;` executes

The output of the statements:

```
System.out.println(illusObject1);
System.out.println(illusObject2);
```

is:

```
x = 3, y = 1, count = 2
x = 5, y = 1, count = 2
```

The program in Example 8-6 further illustrates how **`static`** members of a class work.

EXAMPLE 8-6

```
public class StaticMembers
{
    public static void main(String[] args)
    {
        Illustrate illusObject1 = new Illustrate(3);   //Line 1
        Illustrate illusObject2 = new Illustrate(5);   //Line 2

        Illustrate.incrementY();                       //Line 3
        Illustrate.count++;                            //Line 4

        System.out.println("Line 5: illusObject1: "
                           + illusObject1);            //Line 5
        System.out.println("Line 6: illusObject2: "
                           + illusObject2);            //Line 6

        System.out.println("Line 7: ***Increment y "
                          + "using illusObject1***");  //Line 7
        illusObject1.incrementY();                     //Line 8

        illusObject1.setX(8);                          //Line 9

        System.out.println("Line 10: illusObject1: "
                           + illusObject1);            //Line 10
```

```
        System.out.println("Line 11: illusObject2: "
                           + illusObject2);              //Line 11

        System.out.println("Line 12: ***Increment y "
                           + "using illusObject2***");   //Line 12
        illusObject2.incrementY();                       //Line 13

        illusObject2.setX(23);                           //Line 14

        System.out.println("Line 15: illusObject1: "
                           + illusObject1);              //Line 15
        System.out.println("Line 16: illusObject2: "
                           + illusObject2);              //Line 16
    }
}
```

Sample Run:

```
Line 5: illusObject1: x = 3, y = 1, count = 1
Line 6: illusObject2: x = 5, y = 1, count = 1
Line 7: ***Increment y using illusObject1***
Line 10: illusObject1: x = 8, y = 2, count = 1
Line 11: illusObject2: x = 5, y = 2, count = 1
Line 12: ***Increment y using illusObject2***
Line 15: illusObject1: x = 8, y = 3, count = 1
Line 16: illusObject2: x = 23, y = 3, count = 1
```

The preceding program works as follows: The `static` data members `y` and `count` are initialized to `0`. The statements in Lines 1 and 2 create the `Illustrate` objects `illusObject1` and `illusObject2`. The instance variable `x` of `illusObject1` is initialized to `3`; the instance variable `x` of `illusObject2` is initialized to `5`.

The statement in Line 3 uses the name of the `class` `Illustrate` and the method `incrementY` to increment `y`. Because `count` is a `public static` member of the `class` `Illustrate`, the statement in Line 4 uses the name of the `class` `Illustrate` to directly access `count`, and increments it by `1`. The statements in Lines 5 and 6 output the data stored in the objects `illusObject1` and `illusObject2`. Note that the value of `y` for both objects is the same. Similarly, the value of `count` for both objects is the same.

The statement in Line 7 is an output statement. The statement in Line 8 uses the object `illusObject1` and the method `incrementY` to increment `y`. The statement in Line 9 sets the value of the instance variable `x` of `illusObject1` to `8`. Lines 10 and 11 output the data stored in the objects `illusObject1` and `illusObject2`. Note that the value of `y` for both objects is the same. Similarly, the value of `count` for both objects is the same. Moreover, notice that the statement in Line 9 only changes the value of the instance variable `x` of `illusObject1` because `x` is *not* a `static` member of the `class` `Illustrate`.

The statement in Line 13 uses the object `illusObject2` and the method `incrementY` to increment `y`. The statement in Line 14 sets the value of the instance variable `x` of `illusObject2` to `23`. Lines 15 and 16 output the data stored in the objects `illusObject1` and `illusObject2`. Notice that the value of `y` for both objects is the

same. Similarly, the value of count for both objects is the same. Note that the statement in Line 14 only changes the value of the instance variable x of illusObject2 because x is *not* a static member of the class Illustrate.

NOTE

Here are some additional comments on static members of a class. As you have seen in this section, a static method of a class does not need any object to be invoked. It can be called using the name of the class and the dot operator. Therefore, a static method cannot use anything that depends on a calling object. In other words, in the definition of a static method, you cannot use a non-static data member or a non-static method, unless there is a locally declared object that accesses the non-static data member or the non-static method.

Finalizers

Like constructors, **finalizers** are also special types of methods. However, a finalizer is a void method. A class can have only one finalizer, and the finalizer cannot have any parameters. The name of the finalizer is finalize. The method finalize automatically executes when the class object goes out of scope. A typical use of a finalizer is to free up the memory allocated by the object of a class.

Accessor and Mutator Methods

Earlier in this chapter, we defined the terms mutator method and accessor method. This section discusses these terms in detail and explains why such methods are needed to construct a class.

Let us look at the methods of the class Clock. The method setTime sets the values of the data members to the values specified by the user. In other words, it alters or modifies the values of the instance variables. Similarly, the methods incrementHours, incrementMinutes, and incrementSeconds also modify the instance variables. However, methods such as getHours, getMinutes, getSeconds, printTime, and equals only access the values of the data members; they *do not* modify the data members. We can, therefore, divide the methods of the class Clock into two categories: methods that modify the data members, and methods that access, but do not modify, the data members.

This is typically true for any class. That is, almost every class has methods that only access and do not modify the data members, called **accessor methods**, and methods that modify the data members, called **mutator methods**.

Accessor Method: A method of a class that only accesses (that is, does not modify) the value(s) of the data member(s).

Mutator Method: A method of a class that modifies the value(s) of one or more data member(s).

Typically, the instance variables of a class are declared **private** so that the user of a class does not have direct access to them. In general, every class has a set of accessor methods to work with the instance variables. If the data members need to be modified, then the class also has a set of mutator methods. Conventionally, mutator methods begin with the word **set** and accessor methods begin with the word **get**. You might wonder why we need both mutator and accessor methods when we can simply make the instance variables **public**. However, look closely, for example, at the mutator method **setTime** of the **class** **Clock**. Before setting the time, it validates the time. On the other hand, if the instance variables are all **public**, then the user of the class can put any values in the instance variables. Similarly, the accessor methods only return the value(s) of an instance variable(s); that is, they do not modify the values. A well-designed class uses **private** instance variables, accessor methods, and (if needed) mutator methods to implement the OOD principle of encapsulation.

Example 8-7 further illustrates how classes are designed and implemented. The **class** **Person** that we create in this example is very useful; we will use this **class** in subsequent chapters.

EXAMPLE 8-7

Two common attributes of a person are the person's first name and last name. The typical operations on a person's name are to set the name and print the name. The following statements define a **class** with these properties (see Figure 8-17).

```
public class Person
{
    private String firstName; //store the first name
    private String lastName;  //store the last name

        //Default constructor;
        //Initialize firstName and lastName to empty string.
        //Postcondition: firstName = ""; lastName = "";
    public Person()
    {
        firstName = "";
        lastName = "";
    }

        //Constructor with parameters
        //Set firstName and lastName according to the parameters.
        //Postcondition: firstName = first; lastName = last;
    public Person(String first, String last)
    {
        setName(first, last);
    }
```

```
        //Method to output the first name and last name
        //in the form firstName lastName
    public String toString()
    {
        return (firstName + " " + lastName);
    }

        //Method to set firstName and lastName according to
        //the parameters
        //Postcondition: firstName = first; lastName = last;
    public void setName(String first, String last)
    {
        firstName = first;
        lastName = last;
    }

        //Method to return the firstName
        //Postcondition: the value of firstName is returned
    public String getFirstName()
    {
        return firstName;
    }

        //Method to return the lastName
        //Postcondition: the value of lastName is returned
    public String getLastName()
    {
        return lastName;
    }
}
```


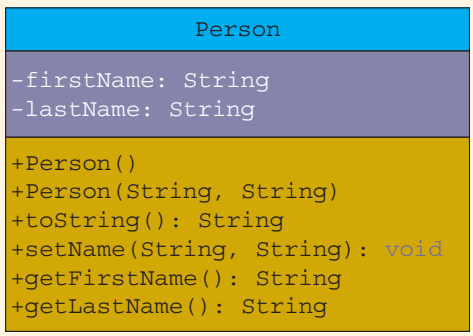

FIGURE 8-17 UML class diagram of the `class Person`

The following program tests the class Person:

```
public class TestProgPerson
{
    public static void main(String[] args)
    {
        Person name = new Person();                          //Line 1

        Person emp = new Person("Donald", "Jackson");        //Line 2

        System.out.println("Line 3: name: " + name);         //Line 3

        name.setName("Ashley", "Blair");                     //Line 4
        System.out.println("Line 5: name: " + name);         //Line 5

        System.out.println("Line 6: emp: " + emp);           //Line 6

        emp.setName("Sandy", "Smith");                       //Line 7
        System.out.println("Line 8: emp: " + emp);           //Line 8
    }//end main
}
```

Sample Run:

```
Line 3: name:
Line 5: name: Ashley Blair
Line 6: emp: Donald Jackson
Line 8: emp: Sandy Smith
```

Debugging—Designing a Class and Documenting the Design

Some beginning programmers mistakenly assume that problem-solving is about coding. These individuals either never become competent programmers, or they come to appreciate the fact that design is a critical step that always precedes coding. By the time we reach the design phase, we already know what the problem is and we focus our attention on how to solve it. Good programmers learn how to solve a problem completely before they write a single line of code. The solution should be understood so thoroughly that, given only pencil, paper, and enough time, the programmer could solve the problem without the use of a computer.

The design is not something that exists only in the programmer's head. It is written down in enough detail that another programmer with the same level of programming skill can take the design and produce the Java code without having to do additional problem-solving.

Let's review the approach we took in designing the class Clock. First we identified the operations that the class needed to perform. We determined that each operation should have its own method and identified the data members required by each of these operations. We determined the type and value, if any, produced and returned by each of these

operations. We determined which members should be **private** and which members should be **public**, and wrote method headings for each member, each beginning with the word **public** or **private**. Following the word **public** or **private**, we stated the return type—except for the constructors, which never have a return type. Then we provided the name of the method followed by parentheses. Any needed parameters, each preceded by its type, were included between the parentheses.

Sometimes, the means by which a method achieves the intended objective is obvious, requiring only a simple statement. Sometimes it takes a great deal of thought to discover the best means of achieving the intended objective, occasionally requiring a more complex statement. Often several means of achieving the objective are considered before one is selected. Determining how to achieve the intended objective should be completed during the design phase, not the coding phase. An **algorithm** describes the means for achieving the intended objective.

An algorithm can be described in many different ways. Algorithms are often described using *pseudocode*. Pseudocode is a mixture of English, Java, and useful symbols, but usually without regard for formal syntax. Whatever form is used to describe the algorithm, it should be sufficiently clear so that a programmer can code the algorithm in Java without having to make any further decisions about how to solve the problem.

The means by which each method achieves its objective is written down as part of the design phase. At one extreme, only a single line is required. At the other extreme, a complicated process with complicated formulas might need to be described. But, in all cases, the means should be clear and complete.

Our design of the **class** `Clock` might look like the following:

```
public class Clock
{
// data members

    private int hr;
    private int min;
    private int sec;

// methods

    public Clock()
    {
        // set time to 0,0,0
    }

    public Clock(int hours, int minutes, int seconds)
    {
        // set time to according to the parameters
    }

    public void setTime(int hours, int minutes, int seconds)
    {
        // set time to according to the parameters

    }
```

```
    public int getHours()
    {
        // return hr
    }

    public int getMinutes()
    {
        // return min
    }

    public int getSeconds()
    {
        //return sec
    }

    public void printTime()
    {
        // print hr:min:sec
    }

    public void incrementSeconds()
    {
        // increment sec by 1
    }

    public void incrementMinutes()
    {
        // increment min by 1
    }

    public void incrementHours()
    {
        // increment hr by 1
    }

    public boolean equals(Clock otherClock)
    {
         // compare this time with the time of otherClock
    }

    public void makeCopy(Clock otherClock)
    {
         // copy time of otherClock
    }

    public Clock getCopy()
    {
         // return a copy of this time
    }
}
```

The current version of the `class Clock` can be coded directly from this design.

Avoid the temptation to skip the design phase. Even though the first programs you write can be made to run by going directly to the coding phase, this approach works only for very small programs. More importantly, it creates a bad habit, which is easy to form but hard to live with—whereas good habits are hard to form but easy to live with. Experience demonstrates consistently that the total time required to produce, debug, and maintain a properly designed program is significantly less than the total time required to produce, debug, and maintain a program with an incomplete design or with no design at all. In fact, programs with incomplete designs, or no designs at all, often never achieve their intended objectives.

Debugging—Design Walk-Throughs

In Chapter 2, you learned about using code walk-throughs to find and remove bugs from programs. The same principles apply to design walk-throughs. Typically, a design walk-through takes place as the design is being finalized and before any code is written.

Except for the syntactic bugs that show up in your program, many of the bugs that you encounter in your programs creep in at design time. Sometimes an important operation is omitted. Sometimes we fail to consider potential future use of the class, and we make a class needlessly specialized. Sometimes too much is expected of a single method, when instead two or more methods should be written to achieve the intended objective. Sometimes not all the data members required by the method are identified and provided. Sometimes the value to be produced and returned by the method is characterized improperly. Sometimes the `public` or `private` status of a member is determined incorrectly. For example, a `public` data member is seldom if ever appropriate. All of these problems can be corrected at design time before a single line of code is written.

Sometimes, in the interest of generality, programmers provide methods that are unlikely to ever be used. This creates excess baggage that can make it difficult to design, implement, test, and maintain programs. Occasionally data members are passed into a method even though the method makes no use of them, or a method returns a value that is never used. All of these excesses make programs more difficult to develop. As a programmer, you should take steps to avoid them.

With a code walk-through, a programmer begins by trying to find and fix these problems himself. The programmer should verify that each intended operation is represented by one or more methods. He should verify that each method receives only the data members needed to achieve its intended objective. The programmer should verify that any intended value is returned by the method. He should also review the `public` or `private` status of each member. Next, the programmer should think through the ranges of data that could be passed to each method. By walking through it in his mind, he verifies that, in every case, the method performs as intended with each variety of data that could be passed to it.

At this point, it may be prudent for the programmer to repeat the design walk-through process with someone who is learning to design programs or who has learned to design programs already. As always, the programmer should be sure that he has prepared his design carefully before presenting it to someone else. In the process of doing so, he may find and correct one or more bugs that he missed during his previous design review. As the programmer explains his design, both he and his audience will have an opportunity to look carefully and methodically at each aspect.

As before, avoid the temptation to shortchange the design phase by "cutting to the chase." Deficiencies encountered at design time are much easier to correct than deficiencies encountered after coding has begun.

Reference `this` (Optional)

In this chapter, we defined the `class` `Clock`. Suppose that `myClock` is a reference variable of type `Clock`. Suppose that the object `myClock` has been created. Consider the following statements:

```
myClock.setTime(5, 6, 59);          //Line 1
myClock.incrementSeconds();         //Line 2
```

The statement in Line 1 uses the method `setTime` to set the instance variables `hr`, `min`, and `sec` of the object `myClock` to `5`, `6`, and `59`, respectively. The statement in Line 2 uses the method `incrementSeconds` to increment the time of the object `myClock` by one second. The statement in Line 2 also results in a call to the method `incrementMinutes` because, after incrementing the value of `sec` by `1`, the value of `sec` becomes `60`, which then is reset to `0`, and the method `incrementMinutes` is invoked.

How do you think Java makes sure that the statement in Line 1 sets the instance variables of the object `myClock` and not of another `Clock` object? How does Java make sure that when the method `incrementSeconds` calls the method `incrementMinutes`, the method `incrementMinutes` increments the value of the instance variable `min` of the object `myClock` and not of another `Clock` object?

The answer is that every object has access to a reference of itself. The name of this reference is `this`. In Java, `this` is a reserved word.

Java implicitly uses the reference `this` to refer to both the instance variables and the methods of a class. Recall that the definition of the method `setTime` is:

```
public void setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;
```

```
    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

In the method `setTime`, the statement:

```
hr = hours;
```

is, in fact, equivalent to the statement:

```
this.hr = hours;
```

In this statement, the reference `this` is used explicitly. You can explicitly use the reference `this` and write the equivalent definition of the method `setTime` as follows:

```
public void setTime(int hr, int min, int sec)
{
    if (0 <= hr && hr < 24)
        this.hr = hr;
    else
        this.hr = 0;

    if (0 <= min && min < 60)
        this.min = min;
    else
        this.min = 0;

    if (0 <= sec && sec < 60)
        this.sec = sec;
    else
        this.sec = 0;
}
```

Notice that in the preceding definition of the method `setTime`, the name of the formal parameters and the name of the instance variables are the same. In this definition of the method `setTime`, the expression `this.hr` means the instance variable `hr`, not the formal parameter `hr`, and so on. Because the code explicitly uses the reference `this`, the compiler can distinguish between the instance variables and the formal parameters. Of course, you could have kept the name of the formal parameters as before and still used the reference `this` as shown in the code.

Similarly, explicitly using the reference this, you can write the definition of the method incrementSeconds as follows:

```
public void incrementSeconds()
{
    this.sec++;

    if (this.sec > 59)
    {
        this.sec = 0;
        this.incrementMinutes(); //increment minutes
    }
}
```

Cascaded Method Calls (Optional)

In addition to explicitly referring to the instance variables and methods of an object, the reference this has another use—to implement cascaded method calls. We explain this with the help of an example.

In Example 8-7, we designed the class Person to implement a person's name in a program. Here, we extend the definition of the class Person to individually set a person's first name and last name, and then return a reference to the object, using this. The following code is the extended definition of the class Person. (The methods setFirstName and setLastName are added to this definition of the class Person.)

```
public class Person
{
    private String firstName; //store the first name
    private String lastName;  //store the last name

        //Default constructor;
        //Initialize firstName and lastName to empty string.
        //Postcondition: firstName = ""; lastName = "";
    public Person()
    {
        firstName = "";
        lastName = "";
    }

        //Constructor with parameters
        //Set firstName and lastName according to the parameters.
        //Postcondition: firstName = first; lastName = last;
    public Person(String first, String last)
    {
        setName(first, last);
    }
```

```
    //Method to return the first name and last name
    //in the form firstName lastName
public String toString()
{
    return (firstName + " " + lastName);
}

    //Method to set firstName and lastName according to
    //the parameters
    //Postcondition: firstName = first; lastName = last;
public void setName(String first, String last)
{
    firstName = first;
    lastName = last;
}

    //Method to set the last name
    //Postcondition: lastName = last;
    //       After setting the last name, a reference
    //       of the object is returned.
public Person setLastName(String last)
{
    lastName = last;

    return this;
}

    //Method to set the first name
    //Postcondition: firstName = first;
    //       After setting the first name, a reference
    //       of the object is returned.
public Person setFirstName(String first)
{
    firstName = first;

    return this;
}

   //Method to return the firstName
   //Postcondition: the value of firstName is returned
public String getFirstName()
{
    return firstName;
}

   //Method to return the lastName
   //Postcondition: the value of lastName is returned
public String getLastName()
{
    return lastName;
}
}
```

Consider the following method `main`:

```
public class CascadedMethodCalls
{
   public static void main(String[] args)
   {
      Person student1 =
                new Person("Angela", "Smith");        //Line 1

      Person student2 = new Person();                 //Line 2

      Person student3 = new Person();                 //Line 3
      System.out.println("Line 4 -- Student 1: "
                         + student1);                 //Line 4

      student2.setFirstName("Shelly").
                            setLastName("Malik");     //Line 5

      System.out.println("Line 6 -- Student 2: "
                         + student2);                 //Line 6

      student3.setFirstName("Chelsea");               //Line 7

      System.out.println("Line 8 -- Student 3: "
                         + student3);                 //Line 8

      student3.setLastName("Tomek");                  //Line 9

      System.out.println("Line 10 -- Student 3: "
                         + student3);                 //Line 10

   }
}
```

Sample Run:

```
Line 4 -- Student 1: Angela Smith
Line 6 -- Student 2: Shelly Malik
Line 8 -- Student 3: Chelsea
Line 10 -- Student 3: Chelsea Tomek
```

The statements in Lines 1, 2, and 3 declare the variables `student1`, `student2`, and `student3` and also instantiate the objects. The instance variables of the objects `student2` and `student3` are initialized to empty strings. The statement in Line 4 outputs the value of `student1`. The statement in Line 5 works as follows. In the statement:

```
student2.setFirstName("Shelly").setLastName("Malik");
```

first the expression:

```
student2.setFirstName("Shelly")
```

is executed because the associativity of the dot operator is from left to right. This expression sets the first name to `"Shelly"` and returns a reference to the object, which is the object `student2`. Thus, the next expression executed is:

```
student2.setLastName("Malik")
```

which sets the last name of the object `student2` to `"Malik"`. The statement in Line 6 outputs the value of `student2`. The statement in Line 7 sets the first name of `student3` to `"Chelsea"`, and the statement in Line 8 outputs `student3`. Notice the output in Line 8. The output shows only the first name, not the last name, because we have not yet set the last name of the object `student3`. The last name of the object `student3` is still empty, which was set by the statement in Line 3 when `student3` was declared. Next, the statement in Line 9 sets the last name of the object `student3`, and the statement in Line 10 outputs `student3`.

Inner Classes

The classes defined thus far in this chapter are said to have file scope, that is, they are contained within a file, but not within another class. In Chapter 6, while designing the `class RectangleProgram`, we defined the `class CalculateButtonHandler` to handle an action event. The definition of the `class CalculateButtonHandler` is contained within the `class RectangleProgram`. Classes that are defined within other classes are called **inner classes**.

An inner class can be either a complete class definition, such as the `class CalculateButtonHandler`, or an anonymous inner class definition. Anonymous classes are classes with no name.

One of the main uses of inner classes is to handle events—as we did in Chapter 6. A full discussion of inner classes is beyond the scope of this book. In this book, our main use of inner classes is to handle events in a GUI program. For example, see the programming example in Chapter 6 and the GUI part of the programming example in this chapter.

Abstract Data Types

To help you understand an abstract data type (ADT) and how it might be used, we'll provide an analogy. The following items seem unrelated:

- A deck of playing cards
- A set of index cards containing contact information
- Telephone numbers stored in your cellular phone

All three of these items share the following structural properties:

- Each one is a collection of elements.
- There is a first element.

- There is a second element, third element, and so on.
- There is a last element.
- Given an element other than the last element, there is a "next" element.
- Given an element other than the first element, there is a "previous" element.
- An element can be removed from the collection.
- An element can be added to the collection.
- A specified element can be located in the collection by systematically going through the collection.

In your programs, you may want to keep a collection of various elements, such as addresses, students, employees, departments, and projects. This structure commonly appears in various applications, and it is worth studying in its own right. We call this organization a *list*, which is an example of an ADT.

There is a data type called `Vector` (discussed in Chapter 9) with basic operations such as:

- Insert an item.
- Delete an item.
- Find an item.

You can use a `Vector` object to create an address book. You would not need to write a program to insert an address, delete an address, or find an item in your address book. Java also allows you to create your own abstract data types through classes.

An ADT is an abstraction of a commonly appearing data structure, along with a set of defined operations on the data structure.

Abstract data type (ADT): A data type that specifies the logical properties without concern for the implementation details.

Historically, the concept of ADT in computer programming developed as a way of abstracting the common data structure and the associated operations. Along the way, ADT provided **information hiding**. That is, ADT *hides* the implementation details of the operations and the data from the users of the ADT. Users can use the operations of an ADT without knowing how the operation is implemented.

PROGRAMMING EXAMPLE: Candy Machine

A new candy machine is bought for the cafeteria and a program is needed to make the machine function properly. The machine sells candies, chips, gum, and cookies. In this programming example, we write a program to create a Java application program for the candy machine so that it can be put into operation.

We implement this program in two ways. First, we show how to design a non-GUI application program. Then, we show how to design an application program that will create a GUI to make the candy machine operational.

The non-GUI application program should do the following:

1. Show the customer the different products sold by the candy machine.
2. Let the customer make the selection.
3. Show the customer the cost of the item selected.
4. Accept the money from the customer.
5. Release the item.

Input: The item selection and the cost of the item

Output: The selected item

In the next section, we design the candy machine's basic components, which are required by either type of application program—GUI or non-GUI. The difference between the two types is evident when we write the main program to put the candy machine into operation.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

A candy machine has three main components: a built-in cash register, several dispensers to hold and release the products, and the candy machine itself. Therefore, we need to define a class to implement the cash register, a class to implement the dispenser, and a class to implement the candy machine. First, we describe the classes to implement the cash register and dispenser, and then we use these classes to describe the candy machine.

Cash Register

Let's first discuss the properties of a cash register. The register has some cash on hand, it accepts the amount from the customer, and if the amount entered is more than the cost of the item, then—if possible—it returns the change. For simplicity, we assume that the user enters the exact amount for the product. The cash register should also be able to show the candy machine's owner the amount of money in the register at any given time. Let's call the class implementing the cash register `CashRegister`.

The members of the **class** `CashRegister` are listed below and shown in Figure 8-18.

Instance Variables

```
private int cashOnHand;
```

Constructors and Methods

```
public CashRegister()
   //Default constructor
   //To set the cash in the register 500 cents
   //Postcondition: cashOnHand = 500;

public CashRegister(int cashIn)
   //Constructor with parameters
   //Postcondition: cashOnHand = cashIn;

public int currentBalance()
   //Method to show the current amount in the cash register
   //Postcondition: The value of the instance variable
   //               cashOnHand is returned

public void acceptAmount(int amountIn)
   //Method to receive the amount deposited by
   //the customer and update the amount in the register
   //Postcondition: cashOnHand = cashOnHand + amountIn
```

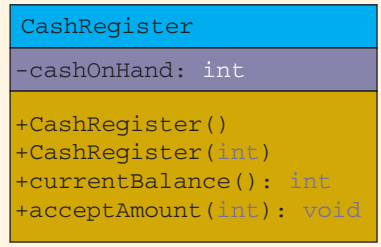

FIGURE 8-18 UML class diagram of the **class** `CashRegister`

Next, we give the definitions of the methods to implement the operations of the **class** `CashRegister`. The definitions of these methods are simple and easy to follow.

The method `currentBalance` shows the current amount in the cash register. The amount stored in the cash register is in cents. Its definition is:

```
public int currentBalance()
{
    return cashOnHand;
}
```

The method `acceptAmount` accepts the amount entered by the customer. It updates the cash in the register by adding the amount entered by the customer to the previous amount in the cash register. The definition of this method is:

```
public void acceptAmount(int amountIn)
{
    cashOnHand = cashOnHand + amountIn;
}
```

The constructor with the parameter sets the value of the instance variable to the value specified by the user. The value is passed as a parameter to the constructor. The definition of the constructor with the parameter is:

```
public CashRegister(int cashIn)
{
    if (cashIn >= 0)
        cashOnHand = cashIn;
    else
        cashOnHand = 500;
}
```

Note that the definition of the constructor checks for valid values of the parameter `cashIn`. If the value of `cashIn` is less than `0`, the value assigned to the instance variable `cashOnHand` is `500`.

The default constructor sets the value of the instance variable `cashOnHand` to `500` cents. Its definition is:

```
public CashRegister()
{
    cashOnHand = 500;
}
```

Now that we have the definitions of all the methods necessary to implement the operations of the `class` `CashRegister`, we can give the definition of `CashRegister`. Its definition is:

```
//class cashRegister

public class CashRegister
{
    private int cashOnHand;    //variable to store the cash
                               //in the register

        //Default constructor to set the cash
        //in the register to 500 cents
        //Postcondition: cashOnHand = 500
    public CashRegister()
    {
        cashOnHand = 500;
    }
```

```
        //Constructor with parameters to set the cash in
        //the register to a specific amount
        //Postcondition: cashOnHand = cashIn
    public CashRegister(int cashIn)
    {
       if (cashIn >= 0)
             cashOnHand = cashIn;
       else
          cashOnHand = 500;
    }

        //Method to show the current amount in the cash register
        //Postcondition: The value of the instance variable
        //               cashOnHand is returned.
    public int currentBalance()
    {
         return cashOnHand;
    }

        //Method to receive the amount deposited by
        //the customer and update the amount in the register
        //Postcondition: cashOnHand = cashOnHand + amountIn
    public void acceptAmount(int amountIn)
    {
         cashOnHand = cashOnHand + amountIn;
    }
}
```

Dispenser

The dispenser releases the selected item if it is not empty. It should show the number of items in the dispenser and the cost of the item. Let's call the class implementing a dispenser `Dispenser`. The members necessary to implement the `class` `Dispenser` are listed next and shown in Figure 8-19.

Instance Variables

```
private int numberOfItems; //variable to store the number of
                           //items in the dispenser

private int cost;  //variable to store the cost of an item
```

Constructors and Methods

```
public Dispenser()
   //Default constructor to set the cost and number of
   //items to the default values
   //Postcondition: numberOfItems = 50; cost = 50;

public Dispenser(int setNoOfItems, int setCost)
   //Constructor with parameters to set the cost and number
   //of items in the dispenser specified by the user
   //Postcondition: numberOfItems = setNoOfItems;
   //               cost = setCost;
```

```
public int getCount()
   //Method to show the number of items in the dispenser
   //Postcondition: The value of the instance variable
   //               numberOfItems is returned

public int getProductCost()
   //Method to show the cost of the item
   //Postcondition: The value of the instance
   //                variable cost is returned

public void makeSale()
   //Method to reduce the number of items by 1
   //Postcondition: numberOfItems = numberOfItems - 1;
```

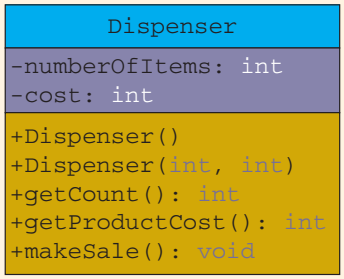

FIGURE 8-19 UML class diagram of the `class` `Dispenser`

Because the candy machine sells four types of items, we will create four objects of type `Dispenser`. The statement:

```
Dispenser chips = new Dispenser(100, 65);
```

creates the object `chips`, sets the number of chip bags in this dispenser to `100`, and sets the cost of each chip bag to `65` cents (see Figure 8-20).

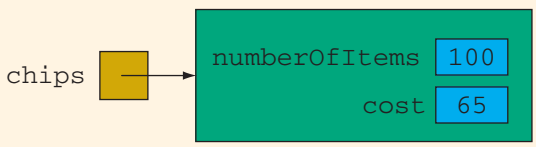

FIGURE 8-20 The object `chips`

Next, we discuss the definitions of the methods to implement the operations of the `class Dispenser`.

The method `getCount` returns the number of items of a particular product. Because the number of items currently in the dispenser is stored in the instance variable `numberOfItems`, the method `getCount` returns the value of the instance variable `numberOfItems`. The definition of this method is:

```
public int getCount()
{
    return numberOfItems;
}
```

The method `getProductCost` returns the cost of a product. Because the cost of a product is stored in the instance variable `cost`, it returns the value of the instance variable `cost`. The definition of this method is:

```
public int getProductCost()
{
    return cost;
}
```

When a product is sold, the number of items in that dispenser is reduced by `1`. Therefore, the method `makeSale` reduces the number of items in the dispenser by `1`. That is, it decrements the value of the instance variable `numberOfItems` by `1`. The definition of this method is:

```
public void makeSale()
{
    numberOfItems--;
}
```

The definition of the constructor checks for valid values of the parameters. If these values are less than `0`, the default values are assigned to the instance variables. The definition of the constructor is:

```
      //constructor with parameters
public Dispenser(int setNoOfItems, int setCost)
{
    if (setNoOfItems >= 0)
       numberOfItems = setNoOfItems;
    else
        numberOfItems = 50;

    if (setCost >= 0)
        cost = setCost;
    else
        cost = 50;
}
```

The default constructor assigns the default values to the instance variables:

```
public Dispenser()
{
    numberOfItems = 50;
    cost = 50;
}
```

The definition of the `class` `Dispenser` is:

```
//class Dispenser

public class Dispenser
{
    private int numberOfItems;   //variable to store the number of
                                 //items in the dispenser
    private int cost;    //variable to store the cost of an item

        //Default constructor to set the cost and number of
        //items to the default values
        //Postcondition: numberOfItems = 50; cost = 50;
    public Dispenser()
    {
        numberOfItems = 50;
        cost = 50;
    }

        //Constructor with parameters to set the cost and number
        //of items in the dispenser specified by the user
        //Postcondition: numberOfItems = setNoOfItems;
        //               cost = setCost;
    public Dispenser(int setNoOfItems, int setCost)
    {
        if (setNoOfItems >= 0)
            numberOfItems = setNoOfItems;
        else
            numberOfItems = 50;

        if (setCost >= 0)
            cost = setCost;
        else
            cost = 50;
    }

        //Method to show the number of items in the dispenser
        //Postcondition: The value of the instance variable
        //                numberOfItems is returned.
    public int getCount()
    {
        return numberOfItems;
    }
```

```
        //Method to show the cost of the item
        //Postcondition: The value of the instance
        //               variable cost is returned.
    public int getProductCost()
    {
        return cost;
    }

        //Method to reduce the number of items by 1
        //Postcondition: numberOfItems = numberOfItems - 1
    public void makeSale()
    {
        numberOfItems--;
    }
}
```

Main Program

When the program executes, it must do the following:

1. Show the different products sold by the candy machine.
2. Show how to select a particular product.
3. Show how to terminate the program.

Furthermore, these instructions must be displayed after processing each selection (except when exiting the program), so that the user need not remember what to do if he or she wants to buy additional items. Once the user makes the appropriate selection, the candy machine must act accordingly. If the user opts to buy an available product, the candy machine should show the cost of the product and ask the user to deposit the money. If the money deposited is at least the cost of the item, the candy machine should sell the item and display an appropriate message.

This discussion translates into the following algorithm:

1. Show the selection to the customer.
2. Get the selection.
3. If the selection is valid and the dispenser corresponding to the selection is not empty, sell the product.

We divide this program into three functions—`showSelection`, `sellProduct`, and `main`.

Method `showSelection`

This method displays the necessary information to help the user select and buy a product. Essentially, it contains the following output statements (we assume that the candy machine sells four types of products):

```
*** Welcome to Shelly's Candy Shop ***"
To select an item, enter
1 for Candy
2 for Chips
```

```
3 for Gum
4 for Cookies
9 to exit
```

The definition of the function `showSelection` is:

```
public static void showSelection()
{
    System.out.println("*** Welcome to Shelly's "
                     + "Candy Shop ***");
    System.out.println("To select an item, enter ");
    System.out.println("1 for Candy");
    System.out.println("2 for Chips");
    System.out.println("3 for Gum");
    System.out.println("4 for Cookies");
    System.out.println("9 to exit");
}//end showSelection
```

Next, we describe the method `sellProduct`.

Method `sellProduct`

This method attempts to sell a particular product selected by the customer. The candy machine contains four dispensers, which correspond to the four products. The first thing this method does is check whether the dispenser holding the product is empty. If the dispenser is empty, the method informs the customer that this product is sold out. If the dispenser is not empty, it tells the user to deposit the necessary amount to buy the product. For simplicity, we assume that this program does not return the extra money deposited by the customer. Therefore, the cash register is updated by adding the money entered by the user.

From this discussion, it follows that the method `sellProduct` must have access to the dispenser holding the product (to decrement the number of items in the dispenser by `1` and to show the cost of the item) as well as access to the cash register (to update the cash). Therefore, this method has two parameters: one corresponding to the dispenser and the other corresponding to the cash register.

In pseudocode, the algorithm for this method is:

1. If the dispenser is not empty
    a. Get the product cost.
    b. Set the variable `coinsRequired` to the price of the product.
    c. Set the variable `coinsInserted` to `0`.
    d. While `coinsRequired` is greater than `0`:
        i. Show and prompt the customer to enter the additional amount.
        ii. Calculate the total amount entered by the customer.
        iii. Determine the amount needed.

e. Update the amount in the cash register.
f. Sell the product—that is, decrement the number of items in the dispenser by 1.
g. Display an appropriate message.

2. If the dispenser is empty, tell the user that this product is sold out.

The definition of the method `sellProduct` is:

```
public static void sellProduct(Dispenser product,
                               CashRegister cRegister)
{
    int price;           //variable to hold the product price
    int coinsInserted;   //variable to hold the amount entered
    int coinsRequired;   //variable to show the extra amount
                         //needed

    if (product.getCount() > 0)                    //Step 1
    {
        price = product.getProductCost();          //Step 1a
        coinsRequired = price;                     //Step 1b
        coinsInserted = 0;                         //Step 1c

        while (coinsRequired > 0)                  //Step 1d
        {
            System.out.print("Please deposit "
                           + coinsRequired
                           + " cents: ");          //Step 1d.i

            coinsInserted = coinsInserted
                          + console.nextInt();     //Step 1d.ii

            coinsRequired = price
                          - coinsInserted;         //Step 1d.iii
        }

        System.out.println();

        cRegister.acceptAmount(coinsInserted);     //Step 1e
        product.makeSale();                        //Step 1f

        System.out.println("Collect your item "
                         + "at the bottom and "
                         + "enjoy.\n");            //Step 1g
    }
    else
        System.out.println("Sorry this item "
                         + "is sold out.\n");      //Step 2
}//end sellProduct
```

Method **main**

The algorithm for the method `main` follows:

1. Create the cash register—that is, create and initialize a `CashRegister` object.
2. Create four dispensers—that is, create and initialize four objects of type `Dispenser`. For example, the statement:

   ```
   Dispenser candy = new Dispenser(100, 50);
   ```

   creates a dispenser object, `candy`, to hold the candies. The number of items in the dispenser is `100`, and the cost of an item is `50` cents.
3. Declare additional variables as necessary.
4. Show the selection; call the method `showSelection`.
5. Get the selection.
6. While not done (a selection of `9` exits the program):
   a. Sell the product; call the method `sellProduct`.
   b. Show the selection; call the method `showSelection`.
   c. Get the selection.

The definition of the method `main` follows:

```
public static void main(String[] args)
{
    CashRegister cashRegister = new CashRegister();  //Step 1
    Dispenser candy = new Dispenser(100, 50);        //Step 2
    Dispenser chips = new Dispenser(100, 65);        //Step 2
    Dispenser gum = new Dispenser(75, 45);           //Step 2
    Dispenser cookies = new Dispenser(100, 85);      //Step 2

    int choice;  //variable to hold the selection    //Step 3

    showSelection();                                 //Step 4
    choice = console.nextInt();                      //Step 5

    while (choice != 9)                              //Step 6
    {
        switch (choice)                              //Step 6a
        {
        case 1:
            sellProduct(candy, cashRegister);
            break;

        case 2:
            sellProduct(chips, cashRegister);
            break;
```

```
        case 3:
            sellProduct(gum, cashRegister);
            break;

        case 4:
            sellProduct(cookies, cashRegister);
            break;

        default:
            System.out.println("Invalid Selection");
        }//end switch

        showSelection();                                    //Step 6b
        choice = console.nextInt();                         //Step 6c
    }//end while
}//end main
```

MAIN PROGRAM LISTING

```
//Program: Candy Machine

import java.util.*;

public class CandyMachine
{
    static Scanner console = new Scanner(System.in);

    //Place the definition of the method main as given above here.

    //Place the definition of the method showSelection as
    //given above here.

    //Place the definition of the method sellProduct as
    //given above here.
}
```

Sample Run: (In this sample run, the user input is shaded.)

```
*** Welcome to Shelly's Candy Shop ***
To select an item, enter
1 for Candy
2 for Chips
3 for Gum
4 for Cookies
9 to exit
1
Please deposit 50 cents: 50

Collect your item at the bottom and enjoy.
```

```
*** Welcome to Shelly's Candy Shop ***
To select an item, enter
1 for Candy
2 for Chips
3 for Gum
4 for Cookies
9 to exit
3
Please deposit 45 cents: 45
Collect your item at the bottom and enjoy.
*** Welcome to Shelly's Candy Shop ***
To select an item, enter
1 for Candy
2 for Chips
3 for Gum
4 for Cookies
9 to exit
9
```

CANDY MACHINE: CREATING A GUI

NOTE If you skipped the GUI part of Chapter 6, you can skip this section.

We will now design an application program that creates the GUI shown in Figure 8-21.

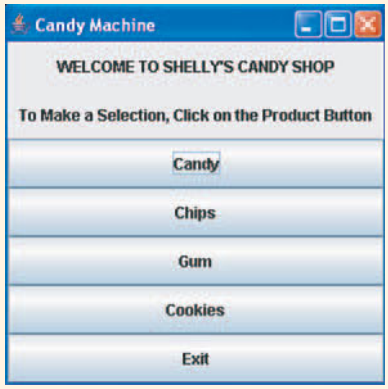

FIGURE 8-21 GUI for the candy machine

The program should do the following:

1. Show the customer the above GUI.
2. Let the customer make the selection.
3. When the user clicks on a product, show the customer its cost, and prompt the customer to enter the money for the product using an input dialog box, as shown in Figure 8-22.

FIGURE 8-22 Input dialog box to enter money for the candy machine

4. Accept the money from the customer.
5. Make the sale and display a dialog box, as shown in Figure 8-23.

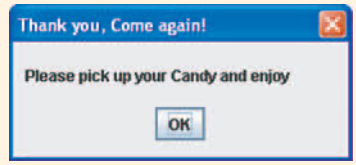

FIGURE 8-23 Output dialog box to show the output of the candy machine

In the first part of this programming example, we designed and implemented the `class`es `CashRegister` and `Dispenser`. Our final step is to revise the main program of the first part to create a GUI.

MAIN PROGRAM

We now describe how to create the candy machine using the `class`es `CashRegister` and `Dispenser` and the GUI components. When the program executes, it must display the GUI shown earlier in Figure 8-21.

The GUI contains a window, two labels, and five buttons. The labels and buttons are placed in the content pane of the window. As you learned in Chapter 6, to create the

window, the application program is created by extending the definition of the class JFrame. Thus, we need the following GUI components:

```
private JLabel headingMainL;    //label for the first line

private JLabel selectionL;      //label for the second line

private JButton exitB, candyB, chipsB, gumB, cookiesB;
```

The following statements create and instantiate these labels and button objects:

```
headingMainL = new JLabel("WELCOME TO SHELLY'S CANDY SHOP",
                          SwingConstants.CENTER);

selectionL = new JLabel("To Make a Selection, "
                      + "Click on the Product Button",
                        SwingConstants.CENTER);

candyB = new JButton("Candy");

chipsB = new JButton("Chips");

gumB = new JButton("Gum");

cookiesB = new JButton("Cookies");

exitB = new JButton("Exit");
```

These components are to be placed in the content pane of the window. The seven components—labels and buttons—are arranged in seven rows. Therefore, the content pane layout will be a grid of 7 rows and 1 column. The following statements get the content pane and add these components to the content pane:

```
Container pane = getContentPane();
setSize(300, 300);

pane.setLayout(new GridLayout(7,1));

pane.add(headingMainL);
pane.add(selectionL);
pane.add(candyB);
pane.add(chipsB);
pane.add(gumB);
pane.add(cookiesB);
pane.add(exitB);
```

EVENT HANDLING

When the user clicks on a product button, it generates an action event. There are five buttons, each generating an action event. To handle these action events, we use the same process that we used in Chapter 6. That is:

1. Create a class implementing the `interface` `ActionListener`.
2. Provide the definition of the method `actionPerformed`.
3. Create and instantiate an object, action listener, of the class type created in Step 1.
4. Register the listener of Step 3 to each button.

In Chapter 6, we created a separate class for each of the buttons and then created a separate listener for each button. In this new program, rather than create a separate class for each button, we create only one class. Recall that the heading of the method `actionPerformed` is:

```
public void actionPerformed(ActionEvent e)
```

In Chapter 6, while providing the definition of this method, we ignored the formal parameter `e`. The formal parameter `e` is a reference variable of the `ActionEvent` type. The `class` `ActionEvent` contains `getActionCommand` (a method without parameters), which can be used to identify which button generated the event. For example, the expression:

```
e.getActionCommand()
```

returns the string containing the label of the component generating the event. We can now use the appropriate `String` method to determine the button generating the event.

If the user clicks on one of the product buttons, then the candy machine attempts to sell the product. Therefore, the action of clicking on a product button is to sell. For this, we write the method `sellProduct` (discussed later in this programming example). If the user clicks on the `Exit` button, the program should terminate. Let's call the class to handle these events `ButtonHandler`. Its definition is:

```
private class ButtonHandler implements ActionListener
{
    public void actionPerformed (ActionEvent e)
    {
        if (e.getActionCommand().equals("Exit"))
            System.exit(0);
        else if (e.getActionCommand().equals("Candy"))
            sellProduct(candy, "Candy");
        else if (e.getActionCommand().equals("Chips"))
            sellProduct(chips, "Chips");
        else if (e.getActionCommand().equals("Gum"))
            sellProduct(gum, "Gum");
        else if (e.getActionCommand().equals("Cookies"))
            sellProduct(cookies, "Cookies");
    }
}
```

You can now declare, instantiate, and register the listener as follows:

```
private ButtonHandler pbHandler;  //declare the listener

pbHandler = new ButtonHandler();  //instantiate the object

    //register the listener with each button
candyB.addActionListener(pbHandler);
chipsB.addActionListener(pbHandler);
gumB.addActionListener(pbHandler);
cookiesB.addActionListener(pbHandler);
exitB.addActionListener(pbHandler);
```

Next, we describe the method `sellProduct`.

Method `sellProduct`

The definition of this method is similar to the one we designed for the non-GUI program. (We give the definition here for the sake of completeness.) This method attempts to sell a particular product selected by the customer. The candy machine contains four dispensers, which correspond to the four products. These dispensers will be declared as instance variables. Therefore, the dispenser of the product to be sold and the name of the product are passed as parameters to this method. Because the cash register will be declared as an instance variable, this method can directly access the cash register.

This definition of the method `sellProduct` is:

```
private void sellProduct(Dispenser product, String productName)
{
    int coinsInserted = 0;
    int price;
    int coinsRequired;
    String str;

    if (product.getCount() > 0)
    {
        price = product.getProductCost();
        coinsRequired = price - coinsInserted;

        while (coinsRequired > 0)
        {
            str = JOptionPane.showInputDialog("To buy "
                                + productName
                                + " please insert "
                                + coinsRequired + " cents");
```

```
            coinsInserted = coinsInserted
                          + Integer.parseInt(str);
            coinsRequired = price - coinsInserted;
        }

        cashRegister.acceptAmount(coinsInserted);
        product.makeSale();

        JOptionPane.showMessageDialog(null,"Please pick up your "
                                + productName + " and enjoy",
                                 "Thank you, Come again!",
                                JOptionPane.PLAIN_MESSAGE);
    }
    else            //dispenser is empty
        JOptionPane.showMessageDialog(null,"Sorry "
                                + productName
                                + " is sold out\n" +
                                "Make another selection",
                                "Thank you, Come again!",
                                JOptionPane.PLAIN_MESSAGE);
}//end sellProduct
```

We have described the method `sellProduct` and the other necessary components, so next we will write the Java application program for the candy machine.

The algorithm is as follows:

1. Create the cash register—that is, declare a reference variable of type `CashRegister` and instantiate the object.
2. Create four dispensers—that is, declare four reference variables of type `Dispenser` and instantiate the appropriate `Dispenser` objects. For example, the statement:

   ```
   Dispenser candy = new Dispenser(100, 50);
   ```

declares `candy` to be a reference variable of the `Dispenser` type and instantiates the object `candy` to hold the candies. The number of items in the object `candy` is `100`, and the cost of a candy is `50` cents.

3. Create the other objects, such as labels and buttons, as previously described.
4. Display the GUI showing the candy machine, as described at the beginning of this programming example.
5. Get and process the selection.

The complete programming listing is available on the Web site and the CD accompanying this book.

QUICK REVIEW

1. A `class` is a collection of a specific number of components.
2. Components of a `class` are called the members of the class.
3. Members of a `class` are accessed by name.
4. In Java, `class` is a reserved word, and it defines only a data type; no memory is allocated.
5. Members of a class are classified into four categories. The three typically used categories are `private`, `protected`, or `public`.
6. The `private` members of a class are not directly accessible outside the class.
7. The `public` members of a class are accessible outside the class.
8. The `public` members are declared using the modifier `public`.
9. The `private` members are declared using the modifier `private`.
10. A member of a class can be a method, a variable, or an inner class.
11. If any member of a class is a variable, it is declared like any other variable.
12. In Java, a `class` is a definition.
13. Non-`static` variables of a `class` are called instance variables of that `class`.
14. Non-`static` methods of a class are called instance methods.
15. Constructors permit the data members to be initialized when an object is declared.
16. The name of a constructor is the same as the name of the class.
17. A class can have more than one constructor.
18. A constructor without parameters is called the default constructor.
19. Constructors automatically execute when a class object is created.
20. In a UML class diagram, the top box contains the name of the class. The middle box contains the data members and their data types. The bottom box contains the methods' names, parameter list, and return type. A `+` (plus) sign in front of a member indicates that the member is a `public` member; a `-` (minus) sign indicates that this is a `private` member. The `#` symbol before a member name indicates that the member is a `protected` member.
21. In shallow copying, two or more reference variables of the same type refer to the same object.
22. In deep copying, each reference variable refers to its own object.
23. A reference variable follows the same scope rules as other variables.
24. A member of a class is local to the class.
25. You access a `public class` member outside the `class` through the reference variable name or the `class` name (for `static` members) and the member access operator (`.`).

26. The copy constructor executes when an object is instantiated and initialized using an existing object.
27. The method `toString` is a `public` value-returning method. It does not take any parameters and returns the address of a `String` object.
28. The methods `print`, `println`, and `printf` output the string created by the method `toString`.
29. The default definition of the method `toString` creates a `String` that is the name of the object's `class` name followed by the object's hash code.
30. The modifier `static` in the heading of the method of a class specifies that the method can be invoked by using the name of the class.
31. If a data member of a class is declared using the modifier `static`, that data member can be invoked by using the name of the class.
32. `static` data members of a `class` *exist* even when no object of the `class` type is instantiated. Moreover, `static` variables are initialized to their default values.
33. Finalizers automatically execute when a class object goes out of scope.
34. A `class` can have only one finalizer, and the finalizer has no parameters.
35. The name of the finalizer is `finalize`.
36. A method of a class that only accesses (that is, does not modify) the value(s) of the data member(s) is called an accessor method.
37. A method of a class that modifies the value(s) of the data member(s) is called a mutator method.
38. Java implicitly uses the reference `this` to refer to both the instance variables and the methods of a class.
39. Classes that are defined within another class are called inner classes.
40. A data type that specifies the logical properties without the implementation details is called an abstract data type (ADT).

EXERCISES

1. Mark the following statements as true or false:
   a. The instance variables of a class must be of the same type.
   b. The methods of a class must be `public`.
   c. A class can have more than one constructor.
   d. A constructor can return a value of the `int` type.
   e. An accessor method of a class accesses and modifies the data members of the class.

2. Find the syntax errors in the definitions of the following classes:

a.
```
public class AA
{
    private int x;
    private int y;

    public void print()
    {
        System.out.println(x + " " + y);
    }
    public int sum()
    {
        return x + y;
    }

    public AA()
    {
        x = 0;
        y = 0;
    }

    public int AA(int a, int b)
    {
        x = a;
        y = b;
    }
}
```

b.
```
public class BB
{
    private int one;
    private int two;

    public boolean equal()
    {
        return (one == two);
    }

    public print()
    {
        System.out.println(one + " " + two);
    }

    public BB(int a, int b)
    {
        one = a;
        two = b;
    }
}
```

3. Consider the definition of the following class:

```
class CC
{
    private int u;
    private int v;
    private double w;

    public CC()                              //Line 1
    {
    }

    public CC(int a)                         //Line 2
    {
    }

    public CC(int a, int b)                  //Line 3
    {
    }

    public CC(int a, int b, double d)        //Line 4
    {
    }
}
```

   a. Give the line number containing the constructor that is executed in each of the following declarations:

      i. `CC one = new CC();`

      ii. `CC two = new CC(5, 6);`

      iii. `CC three = new CC(2, 8, 3.5);`

   b. Write the definition of the constructor in Line 1 so that the instance variables are initialized to `0`.

   c. Write the definition of the constructor in Line 2 so that the instance variable `u` is initialized according to the value of the parameter, and the instance variables `v` and `w` are initialized to `0`.

   d. Write the definition of the constructor in Line 3 so that the instance variables `u` and `v` are initialized according to the values of the parameters `a` and `b`, respectively, and the instance variable `w` is initialized to `0.0`.

   e. Write the definitions of the constructors in Line 4 so that the instance variables `u`, `v`, and `w` are initialized according to the values of the parameters `a`, `b`, and `d`, respectively.

4. Write a Java statement that creates the object `mysteryClock` of the `Clock` type and initializes the instance variables `hr`, `min`, and `sec` of `mysteryClock` to `7`, `18`, and `39`, respectively.

5. Given the statements:

```
Clock firstClock = new Clock(2, 6, 35);
Clock secondClock = new Clock(6, 23, 17);
firstClock = secondClock;
```

what is the output of the following statements?

```
firstClock.print();
System.out.println();
secondClock.print();
System.out.println();
```

6. Consider the following declarations:

```
public class XClass
{
    private int u;
    private double w;

    public XClass()
    {
    }

    public XClass(int a, double b)
    {
    }

    public void func()
    {
    }

    public void print()
    {
    }
}

XClass x = new XClass(10, 20.75);
```

a. How many members does `class XClass` have?

b. How many `private` members does `class XClass` have?

c. How many constructors does `class XClass` have?

d. Write the definition of the member `func` so that `u` is set to `10` and `w` is set to `15.3`.

e. Write the definition of the member `print` that prints the contents of `u` and `w`.

f. Write the definition of the default constructor of the `class XClass` so that the instance variables are initialized to `0`.

g. Write the definition of the constructor with parameters of the `class XClass` so that the instance variable `u` is initialized to the value of `a` and the instance variable `w` is initialized to the value of `b`.

h. Write a Java statement that prints the values of the instance variables of `x`.

i. Write a Java statement that creates the `XClass` object `t` and initializes the instance variables of `t` to `20` and `35.0`, respectively.

7. Explain shallow copying.
8. Explain deep copying.
9. Suppose that two reference variables, say `aa` and `bb`, of the same type point to two different objects. What happens when you use the assignment operator to copy the value of `aa` into `bb`?
10. Assume that the method `toString` is defined for the `class` `Clock` as given in this chapter. What is the output of the following statements?

```
Clock firstClock;
Clock secondClock = new Clock(6, 23, 17);

firstClock = secondClock.getCopy();

System.out.println(firstClock);
```

11. What is the purpose of the copy constructor?
12. How does Java use the reference `this`?
13. Can you use the relational operator == to determine whether two different objects of the same `class` type contain the same data?
14. Consider the definition of the following `class`:

```
class TestClass
{
    private int x;
    private int y;

      //Default constructor to initialize
      //the instance variables to 0
    public TestClass()
    {
    }

      //Constructors with parameters to initialize the
      //instance variables to the values specified by
      //the parameters
      //Postcondition: x = a; y = b;
    TestClass(int a, int b)
    {
    }

      //return the sum of the instance variables
    public int sum()
    {
    }

      //print the values of the instance variables
    public void print()
    {
    }
}
```

a. Write the definitions of the methods as described in the definition of the class TestClass.

b. Write a test program to test various operations of the class TestClass.

15. Write the definition of a class that has the following properties:

a. The name of the class is Secret.

b. The class Secret has four instance variables: name of type String, age and weight of type int, and height of type double.

c. The class Secret has the following methods:

print—outputs the data stored in the data members with the appropriate titles

setName—method to set the name

setAge—method to set the age

setWeight—method to set the weight

setHeight—method to set the height

getName—value-returning method to return the name

getAge—value-returning method to return the age

getWeight—value-returning method to return the weight

getHeight—value-returning method to return the height

default constructor—the default value of name is the empty string ""; the default values of age, weight, and height are 0

constructor with parameters—sets the values of the instance variables name, age, weight, and height to the values specified by the user

d. Write the definitions of the method members of the class Secret, as described in part c.

16. Consider the following definition of the class MyClass:

```
class MyClass
{
    private int x;
    private static int count;

       //default constructor
       //Postcondition: x = 0
    public MyClass()
    {
        //write the definition
    }

       //constructor with a parameter
       //Postcondition: x = a
    public MyClass(int a)
```

```
    {
        //write the definition
    }

       //Method to set the value of x
       //Postcondition: x = a
    public void setX(int a);
    {
        //write the definition
    }

       //Method to output x.
    public void printX()
    {
        //write the definition
    }

       //Method to output count
    public static void printCount()
    {
        //write the definition
    }

      //Method to increment count
      //Postcondition: count++
    public static int incrementCount()
    {
        //write the definition
    }
}
```

a. Write a Java statement that increments the value of `count` by `1`.

b. Write a Java statement that outputs the value of `count`.

c. Write the definitions of the methods and the constructors of the `class` `MyClass` as described in its definition.

d. Write a Java statement that declares `myObject1` to be a `MyClass` object and initializes its instance variable `x` to `5`.

e. Write a Java statement that declares `myObject2` to be a `MyClass` object and initializes its instance variable `x` to `7`.

f. Which of the following statements are valid? (Assume that `myObject1` and `myObject2` are as declared in parts d and e.)

```
myObject1.printCount();      //Line 1
myObject1.printX();          //Line 2
MyClass.printCount();        //Line 3
MyClass.printX();            //Line 4
MyClass.count++;             //Line 5
```

g. Assume that `myObject1` and `myObject2` are as declared in parts d and e. After you have written the definition of the methods of the `class MyClass`, what is the output of the following Java code?

```
myObject1.printX();
myObject1.incrementCount();
MyClass.incrementCount();
myObject1.printCount();
myObject2.printCount();
myObject2.printX();
myObject1.setX(14);
myObject1.incrementCount();
myObject1.printX();
myObject1.printCount();
myObject2.printCount();
```

PROGRAMMING EXERCISES

1. The `class Clock` given in the chapter only allows the time to be incremented by one second, one minute, or one hour. Rewrite the definition of the `class Clock` by including additional members so that time can also be decremented by one second, one minute, or one hour. Also write a program to test your class.
2. Write a program that converts a number entered in Roman numerals to decimal. Your program should consist of a `class`, say, `Roman`. An object of type `Roman` should do the following:
   a. Store the number as a Roman numeral.
   b. Convert and store the number into decimal.
   c. Print the number as a Roman numeral or decimal number as requested by the user.

      The decimal values of the Roman numerals are:

      ```
      M     1000
      D      500
      C      100
      L       50
      X       10
      V        5
      I        1
      ```

   d. Test your program using the following Roman numerals: `MCXIV`, `CCCLIX`, and `MDCLXVI`.

3. Design and implement the `class` `Day` that implements the day of the week in a program. The `class` `Day` should store the day, such as `Sun` for Sunday. The program should be able to perform the following operations on an object of type `Day`:
   a. Set the day.
   b. Print the day.
   c. Return the day.
   d. Return the next day.
   e. Return the previous day.
   f. Calculate and return the day by adding certain days to the current day. For example, if the current day is Monday and we add four days, the day to be returned is Friday. Similarly, if today is Tuesday and we add 13 days, the day to be returned is Monday.
   g. Add the appropriate constructors.
   h. Write the definitions of the methods to implement the operations for the `class` `Day`, as defined in a through g.
   i. Write a program to test various operations on the `class` `Day`.
4. a. Example 8-7 defined the `class` `Person` to store the name of a person. The methods that we included merely set the name and print the name of a person. Redefine the `class` `Person` so that, in addition to what the existing `class` does, you can:
      i. Set the last name only.
      ii. Set the first name only.
      iii. Set the middle name.
      iv. Check whether a given last name is the same as the last name of this person.
      v. Check whether a given first name is the same as the first name of this person.
      vi. Check whether a given middle name is the same as the middle name of this person.
   b. Add the method `equals` that returns true if two objects contain the same first, middle, and last name.
   c. Add the method `makeCopy` that copies the instance variables of a `Person` object into another `Person` object.
   d. Add the method `getCopy` that creates and returns the address of the object, which is a copy of another `Person` object.
   e. Add the copy constructor.
   f. Write the definitions of the methods of the `class` `Person` to implement the operations for this `class`.
   g. Write a program that tests various operations of the `class` `Person`.

5. Redo Example 7-3, Chapter 7, so that it uses the `class` `RollDie` to roll a die.

6. a. Some of the characteristics of a book are the title, author(s), publisher, ISBN, price, and year of publication. Design the `class` `Book` that defines the book as an ADT.

   Each object of the `class` `Book` can hold the following information about a book: title, up to four authors, publisher, ISBN, price, year of publication, and number of copies in stock. To keep track of the number of authors, add another instance variable.

   Include the methods to perform various operations on the objects of `Book`. For example, the usual operations that can be performed on the title are to show the title, set the title, and check whether a title is the actual title of the book. Similarly, the typical operations that can be performed on the number of copies in stock are to show the number of copies in stock, set the number of copies in stock, update the number of copies in stock, and return the number of copies in stock. Add similar operations for the publisher, ISBN, book price, and authors. Add the appropriate constructors.

   b. Write the definitions of the methods of the `class` `Book`.

   c. Write a program that uses the `class` `Book` and tests various operations on the objects of `class` `Book`.

7. In this exercise, you will design the `class` `Member`.

   a. Each object of `Member` can hold the name of a person, member ID, number of books bought, and amount spent.

   b. Include the methods to perform the various operations on the objects of the `class` `Member`—for example, modify, set, and show a person's name. Similarly, update, modify, and show the number of books bought and the amount spent.

   c. Write the definitions of the methods of the `class` `Member`. Also write a program to test your class.

8. The equation of a line in standard form is $ax + by = c$, where a and b both cannot be zero, and a, b, and c are real numbers. If $b \neq 0$, then $-a / b$ is the slope of the line. If $a = 0$, then it is a horizontal line, and if $b = 0$, then it is a vertical line. The slope of a vertical line is undefined. Two lines are parallel if they have the same slope or both are vertical lines. Two lines are perpendicular if one of the lines is horizontal and another is vertical, or if the product of their slopes is –1. Design the `class` `lineType` to store a line. To store a line, you need to store the values of a (coefficient of x), b (coefficient of y), and c. Your class must contain the following operations:

   a. If a line is nonvertical, then determine its slope.

   b. Determine if two lines are equal. (Two lines $a_1x + b_1y = c_1$ and $a_2x + b_2y = c_2$ are equal if either $a_1 = a_2$, $b_1 = b_2$, and $c_1 = c_2$ or $a_1 = ka_2$, $b_1 = kb_2$, and $c_1 = kc_2$ for some real number k.)

c. Determine if two lines are parallel.

d. Determine if two lines are perpendicular.

e. If two lines are not parallel, then find the point of intersection.

Add appropriate constructors to initialize variables of `lineType`. Also write a program to test your class.

9. Rational fractions are of the form a / b, where a and b are integers and $b \neq 0$. In this exercise, by "fractions" we mean rational fractions. Suppose that a / b and c / d are fractions. Arithmetic operations on fractions are defined by the following rules:

$a / b + c / d = (ad + bc) / bd$
$a / b - c / d = (ad - bc) / bd$
$a / b \times c / d = ac / bd$
$(a / b) / (c / d) = ad / bc$, where $c / d \neq 0$

Fractions are compared as follows: a / b *op* c / d if ad *op* bc, where *op* is any of the relational operations. For example, $a / b < c / d$ if $ad < bc$.

Design the `class Fraction` that can be used to manipulate fractions in a program. Among others, the `class Fraction` must include methods to add, subtract, multiply, and divide fractions. When you add, subtract, multiply, or divide fractions, your answer need not be in the lowest terms. Also, override the method `toString` so that the fractions can be output using the output statement.

Write a Java program that, using the `class Fraction`, performs operations on fractions.

CHAPTER

9

ARRAYS

IN THIS CHAPTER, YOU WILL:

- Learn about arrays
- Explore how to declare and manipulate data in arrays
- Learn about the instance variable `length`
- Understand the meaning of "array index out of bounds"
- Become aware of how the assignment and relational operators work with array names
- Discover how to pass an array as a parameter to a method
- Learn how to search an array
- Discover how to manipulate data in a two-dimensional array
- Learn about multidimensional arrays
- Become acquainted with the `class` Vector

In previous chapters, you worked with primitive data types and learned how to construct your own classes. Recall that a variable of a primitive data type can store only one value at a time; on the other hand, a **class** can be defined so that its objects can store more than one value at a time. This chapter introduces a special data structure called an array, which allows the user to group data items of the same type and process them in a convenient way.

Why Do We Need Arrays?

Before we formally define an array, let's consider the following problem. We want to write a Java program that reads five numbers, finds their sum, and prints the numbers in reverse order.

In Chapter 5, you learned how to read numbers, print them, and find their sum. What's different here is that we want to print the numbers in reverse order. We cannot print the first four numbers until we have printed the fifth, and so on. This means that we need to store all the numbers before we can print them in reverse order. From what we have learned so far, the following program accomplishes this task:

```
//Program to read five numbers, find their sum, and print the
//numbers in the reverse order.

import java.util.*;

public class ReversePrintI
{
    static Scanner console = new Scanner(System.in);

    public static void main(String[] args)
    {
        int item0, item1, item2, item3, item4;
        int sum;

        System.out.println("Enter five integers: ");
        item0 = console.nextInt();
        item1 = console.nextInt();
        item2 = console.nextInt();
        item3 = console.nextInt();
        item4 = console.nextInt();

        sum = item0 + item1 + item2 + item3 + item4;

        System.out.println("The sum of the numbers = " + sum);
        System.out.print("The numbers in reverse order are: ");
        System.out.println(item4 + " " + item3 + " " + item2
                           + " " + item1 + " " + item0);
    }
}
```

This program works fine. However, to read 100 (or more) numbers and print them in reverse order, you would have to declare 100 or more variables and write many input and output statements. Thus, for large amounts of data, this type of program is not desirable.

Note the following in the preceding program:

1. Five variables must be declared because the numbers are to be printed in reverse order.
2. All variables are of type `int`—that is, of the same data type.
3. The way in which these variables are declared indicates that the variables to store these numbers have the same name except for the last character, which is a number.

From 1, it follows that you have to declare five variables. From 3, it follows that it would be convenient if you could somehow put the last character, which is a number, into a counter variable and use one `for` loop to count from `0` to `4` for reading, and use another `for` loop to count from `4` to `0` for printing. Finally, because all the variables are of the same type, you should be able to specify how many variables must be declared—as well as their data type—with a simpler statement than the one used previously.

The data structure that lets you do all of these things in Java is called an array.

Arrays

An **array** is a collection (sequence) of a fixed number of variables called **elements** or **components**, wherein all the elements are of the same data type. A **one-dimensional array** is an array in which the elements are arranged in a list form. The remainder of this section discusses one-dimensional arrays. Arrays of two or more dimensions are discussed later in this chapter.

The general form to declare a one-dimensional array is:

```
dataType[] arrayName;    //Line 1
```

where `dataType` is the element type.

In Java, an array is an object, just like the objects discussed in Chapter 8. Because an array is an object, `arrayName` is a reference variable. Therefore, the preceding statement only declares a reference variable. Before we can store the data, we must instantiate the array object.

The general syntax to instantiate an array object is:

```
arrayName = new dataType[intExp];    //Line 2
```

where `intExp` is any expression that evaluates to a positive integer. Also, the value of `intExp` specifies the number of elements in the array.

You can combine the statements in Lines 1 and 2 into one statement as follows:

```
dataType[] arrayName = new dataType[intExp];    //Line 3
```

We typically use statements similar to the one in Line 3 to create arrays to manipulate data.

NOTE When an array is instantiated, Java automatically initializes its elements to their default values. For example, the elements of numeric arrays are initialized to 0, the elements of `char` arrays are initialized to the null character, which is `'\u0000'`, the elements of `boolean` arrays are initialized to `false`.

EXAMPLE 9-1

The statement:

```
int[] num = new int[5];
```

declares and creates the array `num` consisting of `5` elements. Each element is of type `int`. The elements are accessed as `num[0]`, `num[1]`, `num[2]`, `num[3]`, and `num[4]`. Figure 9-1 illustrates the array `num`.

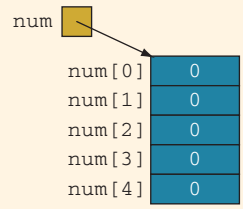

FIGURE 9-1 Array num

NOTE To save space, we also draw an array, as shown in Figure 9-2(a) and 9-2(b).

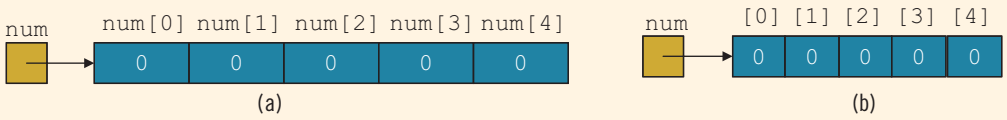

FIGURE 9-2 Array num

Alternate Ways to Declare an Array

Java allows you to declare arrays as follows:

```
int list[];          //Line 1
```

Here, the operator `[]` appears after the identifier `list`, not after the data type `int`.

You should be careful when declaring arrays as in Line 1. Consider the following statements:

```
int alpha[], beta;   //Line 2
int[] gamma, delta;  //Line 3
```

The statement in Line 2 declares the variables `alpha` and `beta`. Similarly, the statement in Line 3 declares the variables `gamma` and `delta`. However, the statement in Line 2 declares only `alpha` to be an array reference variable, while the variable `beta` is an `int` variable. On the other hand, the statement in Line 3 declares both `gamma` and `delta` to be array reference variables.

Traditionally, Java programmers declare arrays as shown in Line 3. We recommend that you do the same.

Accessing Array Elements

The general form (syntax) used to access an array element is:

```
arrayName[indexExp]
```

where `indexExp`, called the **index**, is an expression whose value is a nonnegative integer less than the size of the array. The index value specifies the position of the element in the array. In Java, the array index starts at `0`.

In Java, `[]` is an operator called the **array subscripting operator**.

Consider the following statement:

```
int[] list = new int[10];
```

This statement declares an array `list` of `10` elements. The elements are `list[0]`, `list[1]`, . . ., `list[9]`. In other words, we have declared 10 variables of type `int` (see Figure 9-3).

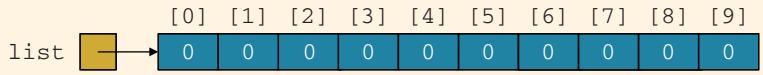

FIGURE 9-3 Array `list`

The assignment statement:

```
list[5] = 34;
```

stores `34` into `list[5]`, which is the sixth element of the array `list` (see Figure 9-4).

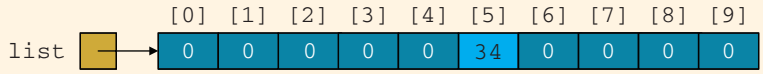

FIGURE 9-4 Array `list` after the execution of the statement `list[5]= 34;`

Suppose `i` is an `int` variable. Then, the assignment statement:

```
list[3] = 63;
```

is equivalent to the assignment statements:

```
i = 3;
list[i] = 63;
```

If `i` is `4`, then the assignment statement:

```
list[2 * i - 3] = 58;
```

stores `58` into `list[5]`, because `2 * i - 3` evaluates to `5`. The index expression is evaluated first, giving the position of the element in the array.

Next, consider the following statements:

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

The first statement stores `10` into `list[3]`, the second statement stores `35` into `list[6]`, and the third statement adds the contents of `list[3]` and `list[6]` and stores the result into `list[5]` (see Figure 9-5).

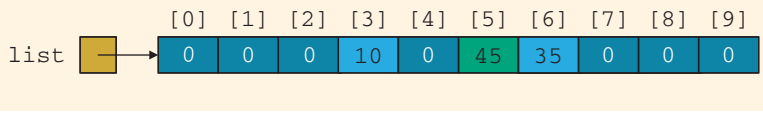

FIGURE 9-5 Array `list` after the execution of the statements `list[3]= 10;`, `list[6]= 35;`, and `list[5] = list[3] + list[6];`

EXAMPLE 9-2

You can also declare arrays as follows:

```
final int ARRAY_SIZE = 10;
int[] list = new int[ARRAY_SIZE];
```

That is, you can first declare a named constant of an integral type, such as `int`, and then use the value of the named constant to specify the size of the array.

Specifying Array Size during Program Execution

When you include a statement in a program to instantiate an array object, it is not necessary to know the size of the array at compile time. During program execution, you can first prompt the user to specify the size of the array and then instantiate the object. The following statements illustrate this concept (suppose that `console` is a `Scanner` object initialized to the standard input device):

```
int arraySize;                                          //Line 1

System.out.print("Enter the size of the array: ");     //Line 2
arraySize = console.nextInt();                          //Line 3
System.out.println();                                   //Line 4

int[] list = new int[arraySize];                        //Line 5
```

The statement in Line 2 asks the user to enter the size of the array when the program executes. The statement in Line 3 inputs the size of the array into `arraySize`. During program execution, the system uses the value of the variable `arraySize` to instantiate the object `list`. For example, if the value of `arraySize` is `15`, `list` is an array of size `15`.

Array Initialization during Declaration

Like any other primitive data type variable, an array can also be initialized with specific values when it is declared. For example, the following Java statement declares an array, `sales`, of five elements and initializes those elements to specific values:

```
double[] sales = {12.25, 32.50, 16.90, 23, 45.68};
```

The **initializer list** contains values, called **initial values**, that are placed between braces and separated by commas. Here, `sales[0] = 12.25`, `sales[1] = 32.50`, `sales[2] = 16.90`, `sales[3] = 23.00`, and `sales[4]= 45.68`.

Note the following about declaring and initializing arrays:

- When declaring and initializing arrays, the size of the array is determined by the number of initial values in the initializer list within the braces.
- If an array is declared and initialized simultaneously, we *do not* use the operator `new` to instantiate the array object.

Arrays and the Instance Variable `length`

Recall that an array is an object; therefore, to store data, the array object must be instantiated. Associated with each array that has been instantiated (that is, for which memory has been allocated to store data), there is a `public` (`final`) instance variable `length`. The variable `length` contains the size of the array. Because `length` is a `public` member, it can be directly accessed in a program using the array name and the dot operator.

Consider the following declaration:

```
int[] list = {10, 20, 30, 40, 50, 60};
```

This statement creates the array `list` of six elements and initializes the elements using the values given. Here, `list.length` is `6`.

Consider the following statement:

```
int[] numList = new int[10];
```

This statement creates the array `numList` of `10` elements and initializes each element to `0`. Because the number of elements of `numList` is `10`, the value of `numList.length` is `10`. Now consider the following statements:

```
numList[0] = 5;
numList[1] = 10;
numList[2] = 15;
numList[3] = 20;
```

These statements store `5`, `10`, `15`, and `20`, respectively, in the first four elements of `numList`. Even though we put data into only the first four elements, the value of `numList.length` is `10`, the total number of array elements.