

Using Methods, Classes, and Objects

In this chapter, you will:

- ① Learn about method calls and placement
- ① Identify the parts of a method
- ① Add parameters to methods
- ① Create methods that return values
- ① Learn about classes and objects
- ① Create a class
- ① Create instance methods in a class
- ① Declare objects and use their methods
- ① Create constructors
- ① Appreciate classes as data types

Understanding Method Calls and Placement

A **method** is a program module that contains a series of statements that carry out a task. You have already seen Java classes that contain a `main()` method, which executes automatically when you run a program. A program's `main()` method can execute additional methods, and those methods can execute others. Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times.

To execute a method, you **invoke** or **call** it. In other words, a **calling method** makes a **method call**, and the method call invokes a **called method**. The calling method is also known as a **client method** because a called method provides a service for its client.

Consider the simple `First` class that you saw in Chapter 1; it displayed a single line of output, "First Java application." Suppose that you want to add three lines of output to this application to display your company's name and address. One approach would be to simply add three new `println()` statements, as shown in the shaded statements in Figure 3-1.

```
public class First
{
    public static void main(String[] args)
    {
        System.out.println("XYZ Company");
        System.out.println("8900 U.S. Hwy 14");
        System.out.println("Crystal Lake, IL 60014");
        System.out.println("First Java application");
    }
}
```

Figure 3-1 The `First` class

Instead of adding the three `println()` statements to the application in Figure 3-1, you might prefer to call a method that executes the three statements. Then the program would look like the one in Figure 3-2. The shaded line contains the call to the `nameAndAddress()` method.

```
public class First
{
    public static void main(String[] args)
    {
        nameAndAddress();
        System.out.println("First Java application");
    }
}
```

Figure 3-2 The `First` class with a call to the `nameAndAddress()` method

There are two major advantages to creating a separate method to display the three address lines. First, the `main()` method remains short and easy to follow because `main()` contains just one statement to call the method, rather than three separate `println()` statements to perform the work of the method. What is more important is that a method is easily reusable. After you create the name and address method, you can use it in any application that needs the company's name and address. In other words, you do the work once, and then you can use the method many times.

Besides adding a call to the method in the `First` class, you must actually write the method. You place a method within a class, but it must be outside of any other methods. In other words, you cannot place a method within another method. Figure 3-3 shows the two locations where you can place additional methods within the `First` class—within the curly braces of the class, but outside of (either before or after) any other methods. Methods can never overlap.

The order of methods in a class has no bearing on the order in which the methods are called or executed. No matter where you place it, the `main()` method is always executed first in any Java application, and it might call any other methods in any order and any number of times. The order in which you call methods, and not their physical placement, is what makes a difference in how an application executes.

```
public class First
{
    // You can place additional methods here, before main()
    public static void main(String[] args)
    {
        nameAndAddress();
        System.out.println("First Java application");
    }
    // You can place additional methods here, after main()
}
```

Figure 3-3 Placement of methods within a class

A `main()` method executes automatically when you run a program, but other methods do not execute simply because you place them within a class—they must be called. A class might contain methods that are never called from a particular application, just as some electronic devices might contain features you never use. For example, you might use a DVR to play movies but never to record TV programs, or you might use your microwave oven for popcorn but never to defrost.

Figure 3-4 shows the `First` class with two methods: the `main()` method and the `nameAndAddress()` method placed after `main()`. Figure 3-5 shows the output from the execution of the `First` class in Figure 3-4. When the program executes, the `main()` method first calls the `nameAndAddress()` method, which displays three lines of output. Then `main()` displays the phrase “First Java application”.



Using a method name to contain or encapsulate a series of statements is an example of the feature that programmers call **abstraction**. Consider abstract art, in which the artist tries to capture the essence of an object without focusing on the details. Similarly, when programmers employ abstraction, they use a general method name in a module rather than list all the detailed activities that will be carried out by the method.

The main() method in this class contains two statements. The first one is a call to the nameAndAddress() method.

```
public class First
{
    public static void main(String[] args)
    {
        nameAndAddress();
        System.out.println("First Java application");
    }

    public static void nameAndAddress()
    {
        System.out.println("XYZ Company");
        System.out.println("8900 U.S. Hwy 14");
        System.out.println("Crystal Lake, IL 60014");
    }
}
```

Figure 3-4 First class with main() calling nameAndAddress()

```
C:\Java>java First
XYZ Company
8900 U.S. Hwy 14
Crystal Lake, IL 60014
First Java application
C:\Java>_
```

Figure 3-5 Output of the First application, including the nameAndAddress() method



Watch the video *Methods*.

TWO TRUTHS & A LIE

Understanding Method Calls and Placement

1. Any class can contain an unlimited number of methods.
2. During one program execution, a method might be called any number of times.
3. A method is usually written within another method.

The false statement is #3. A method is written within a class, but not within any other methods.

Understanding Method Construction

Every method must include the two parts featured in Figure 3-6:

- A **method header**—A method's header provides information about how other methods can interact with it. A method header is also called a **declaration**.
- A **method body** between a pair of curly braces—The method body contains the statements that carry out the work of the method. A method's body is called its **implementation**. Technically, a method is not required to contain any statements in its body, but you usually would have no reason to create an empty method in a class. Sometimes, while developing a program, the programmer creates an empty method as a placeholder and fills in the implementation later. An empty method is called a **stub**.

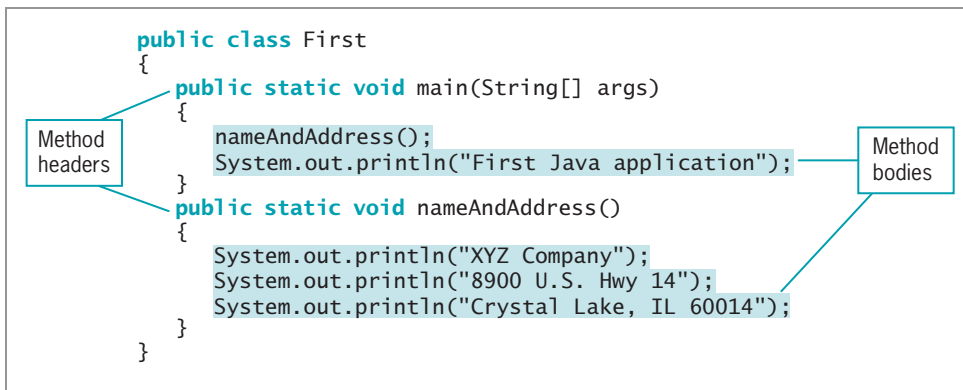


Figure 3-6 The headers and bodies of the methods in the `First` class

The method header is the first line of a method. It contains the following:

- Optional access specifiers
- A return type
- An identifier
- Parentheses

The next few figures compare these parts of a method header for the `main()` method and the `nameAndAddress()` method in the `First` class.

Access Specifiers

Figure 3-7 highlights the optional access specifiers for the two methods in the `First` class. The access specifier for a Java method can be any of the following modifiers: `public`, `private`, `protected`, or, if left unspecified, `package` by default. Most often, methods are given `public` access; this book will cover the other modifiers later. Endowing a method with `public` access means that any other class can use it, not just the class in which the method resides.

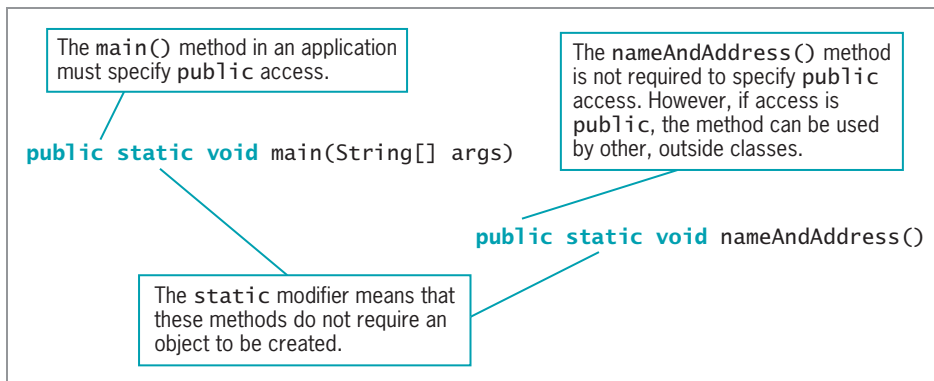


Figure 3-7 Access specifiers for two methods



You first learned the term *access specifier* in Chapter 1. Access specifiers are sometimes called **access modifiers**.

In addition, any method that can be used without instantiating an object requires the keyword modifier `static`. The `main()` method in an application must use the keyword `static`, but other methods, like `nameAndAddress()`, can use it too. You will learn about nonstatic methods later in this chapter.

Return Type

Figure 3-8 features the return types for the `main()` and `nameAndAddress()` methods in the `First` class. A **return type** describes the type of data the method sends back to its calling method. Not all methods **return a value** to their calling methods; a method that returns no data has a return type of `void`. The `main()` method in an application must have a return type of `void`; in this example, `nameAndAddress()` also has a `void` return type. Other methods that you will see later in this chapter have different return types. The phrases *void method* and *method of type void* both refer to a method that has a `void` return type.

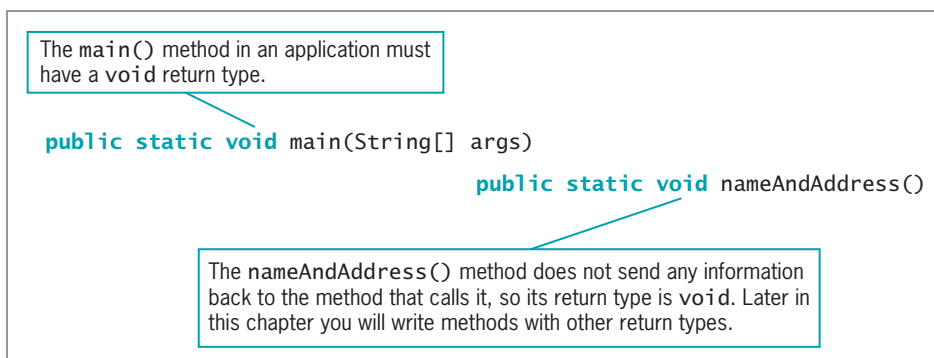


Figure 3-8 Return types for two methods

Method Name

Figure 3-9 highlights the names of the two methods in the `First` class. A method's name can be any legal identifier. That is, like identifiers for classes and variables, a method's identifier must be one word with no embedded spaces, and cannot be a Java keyword. The method that executes first when you run an application must be named `main()`, but you have a lot of leeway in naming other methods that you create within a class. Technically, you could even name another method `main()` as long as you did not include `String[]` within the parentheses, but doing so would be confusing and is not recommended.

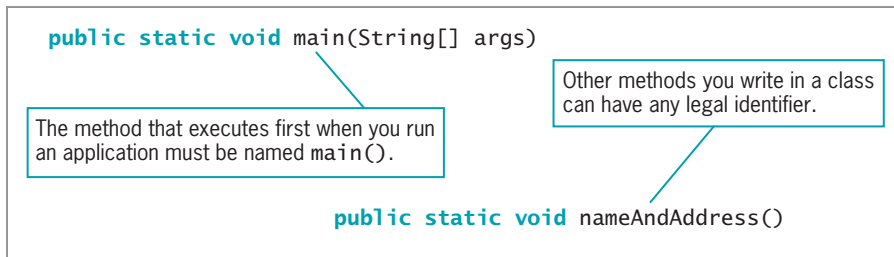


Figure 3-9 Identifiers for two methods

Parentheses

As Figure 3-10 shows, every method header contains a set of parentheses that follow the identifier. The parentheses might contain data to be sent to the method. For example, when you write a `main()` method in a class, the parentheses in its header surround `String[] args`. The `nameAndAddress()` method in the `First` class requires no outside data, so its parentheses are empty. Later in this chapter, you will see several methods that accept data.

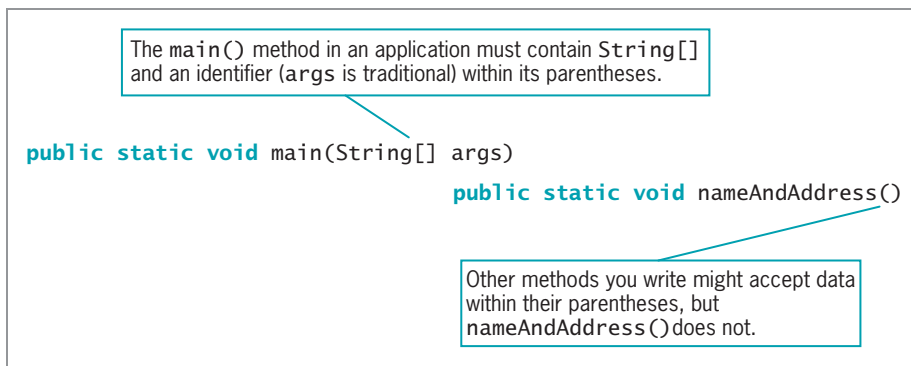


Figure 3-10 Parentheses and their contents for two methods

The full name of the `nameAndAddress()` method is `First.nameAndAddress()`, which includes the class name, a dot, and the method name. (The name does not include an object because `nameAndAddress()` is a `static` method.) A complete name that includes the class is a **fully qualified identifier**. When you use the `nameAndAddress()` method within its

own class, you do not need to use the fully qualified name (although you can); the simple method name alone is enough. However, if you want to use the `nameAndAddress()` method in another class, the compiler does not recognize the method unless you use the full name. You have used similar syntax (including a class name, dot, and method name) when calling the `JOptionPane.showMessageDialog()` method.

Each of two different classes can have its own method named `nameAndAddress()`. Such a method in the second class would be entirely distinct from the identically named method in the first class. You could use both methods in a third class by using their fully qualified identifiers. Two classes in an application cannot have the same name.



Think of the class name as the family name. Within your own family, you might refer to an activity as “the family reunion,” but outside the family people need to use a surname as well, as in “the Anderson family reunion.” Similarly, within a class a method name alone is sufficient, but outside the class you need to use the fully qualified name.

TWO TRUTHS & A LIE

Understanding Method Construction

1. A method header is also called an implementation.
2. When a method is declared with `public` access, methods in other classes can call it.
3. Not all methods return a value, but every method requires a return type.

The false statement is #1. A method header is a declaration; a method body is its implementation.



You Do It

Creating a static Method That Requires No Arguments and Returns No Values

Paradise Day Spa provides many personal services such as haircuts, manicures, and facials. In this section, you create a new class named `ParadiseInfo`, which contains a `main()` method that calls a `displayInfo()` method.

1. Open a new document in your text editor, and type the following shell for the class:

```
public class ParadiseInfo
{
}
```

(continues)

(continued)

- Between the curly braces of the class, indent a few spaces and create the shell for the `main()` method:

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

- Between the braces of the `main()` method, insert a call to the `displayInfo()` method:

```
displayInfo();
```

- Place the `displayInfo()` method outside the `main()` method, just before the closing curly brace for the `ParadiseInfo` class:

```
public static void displayInfo()
{
    System.out.println("Paradise Day Spa wants to pamper you.");
    System.out.println("We will make you look good.");
}
```

- Save the file as **ParadiseInfo.java**.
- Compile the class, and then execute it. The output should look like Figure 3-11.

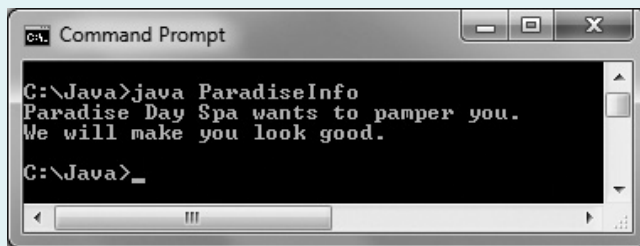


Figure 3-11 Output of the `ParadiseInfo` application

Calling a static Method from Another Class

Next, you see how to call the `displayInfo()` method from a method within another class.

- Open a new document in your text editor, and then enter the following class in which the `main()` method calls the `displayInfo()` method that resides in the `ParadiseInfo` class:

```
public class TestInfo
{
    public static void main(String[] args)
    {
        System.out.println("Calling method from another class:");
        ParadiseInfo.displayInfo();
    }
}
```

(continues)

(continued)

2. Save the file as **TestInfo.java** in the same folder as the `ParadiseInfo` class. If the files are not saved in the same folder and you try to compile the calling class, your compiler issues the error message “cannot find symbol”; the symbol named is the missing class you tried to call.
3. Compile the application and execute it. Your output should look like Figure 3-12. The `TestInfo` class does not contain the `displayInfo()` method; it uses the method from the `ParadiseInfo` class. It’s important that the `displayInfo()` method is `public`. If you had omitted the keyword `public` from the definition of the `displayInfo()` method in the `ParadiseInfo` class, then the `TestInfo` class would not have been able to use it.



```
C:\Java>java TestInfo
Calling method from another class:
Paradise Day Spa wants to pamper you.
We will make you look good.
C:\Java>
```

Figure 3-12 Output of the `TestInfo` application

Examining the Details of a Prewritten static Method

Recall that in Chapter 2, you used the `JOptionPane` class to create statements like the following:

```
JOptionPane.showMessageDialog
(null, "Every bill is due in " + creditDays + " days");
```

In the next steps, you examine the Java API documentation for the `showMessageDialog()` method so that you can better understand how prewritten methods are similar to ones that you write.

1. Using a Web browser, go to the Java Web site and select **Java APIs** and **Java SE 7**.
2. In the alphabetical list of classes, find the **JOptionPane** class and select it.
3. Scroll through the class documentation until you find the **Method Summary**. Then, find the first listed version of the `showMessageDialog()` method. To the left, notice that the method is defined as a `static void` method, just like the `main()` and `displayInfo()` methods discussed earlier in this “You Do It” section. You can use the `static showMessageDialog()` method in your classes by using its class name, a dot, and the method name, in the same way that you used the `ParadiseInfo` `static displayInfo()` method in the outside class named `TestInfo`.

Adding Parameters to Methods

Some methods require that data be sent to them when they are called. Data items you use in a call to a method are called **arguments**. When the method receives the data items, they are called **parameters**. Methods that receive data are flexible because they can produce different results depending on what data they receive.

As a real-life example, when you make a restaurant reservation, you do not need to employ a different method for every date of the year at every possible time of day. Rather, you can supply the date and time as information to the person who carries out the method. The method, recording the reservation, is then carried out in the same manner, no matter what date and time are supplied.

In a program, if you design a method to square numeric values, it makes sense to design a `square()` method that you can supply with an argument that represents the value to be squared, rather than having to develop a `square1()` method (that squares the value 1), a `square2()` method (that squares the value 2), and so on. To call a `square()` method that takes an argument, you might write a statement like `square(17);` or `square(86);`. Similarly, any time it is called, the `println()` method can receive any one of an infinite number of arguments—for example, “Hello”, “Goodbye”, or any other `String`. No matter what message is sent to `println()`, the message is displayed correctly. If the `println()` method could not accept arguments, it would not be practical to use.

An important principle of object-oriented programming is **implementation hiding**, which describes the encapsulation of method details. For example, when you make a real-life restaurant reservation, you do not need to know how the reservation is actually recorded at the restaurant—perhaps it is written in a book, marked on a large chalkboard, or entered into a computerized database. The implementation details don’t concern you as a client, and if the restaurant changes its methods from one year to the next, the change does not affect your use of the reservation method—you still call and provide your name, a date, and a time.

With well-written object-oriented programming methods, using implementation hiding means that a method that calls another must know the name of the called method and what type of information to send. (Usually, you also want to know about any data returned by the method; you will learn about returned data later in this chapter.) However, the client does not need to know how the method works internally. In other words, the calling method needs to understand only the **interface** to the called method. The interface is the only part of a method that the method’s client sees or with which it interacts. In addition, if you substitute a new or revised method implementation, as long as the interface to the method does not change, you won’t need to make any changes in any methods that call the altered method.



Hidden implementation methods are often referred to as existing in a **black box**. Many everyday devices are black boxes—that is, you can use them without understanding how they work. For example, most of us use telephones, television sets, and automobiles without understanding much about their internal mechanisms.

Creating a Method That Receives a Single Parameter

When you write the method declaration for a method that can receive a parameter, you begin by defining the same elements as you do for methods that do not accept parameters—optional access specifiers, the return type for the method, and the method name. In addition, you must include the following items within the method declaration parentheses:

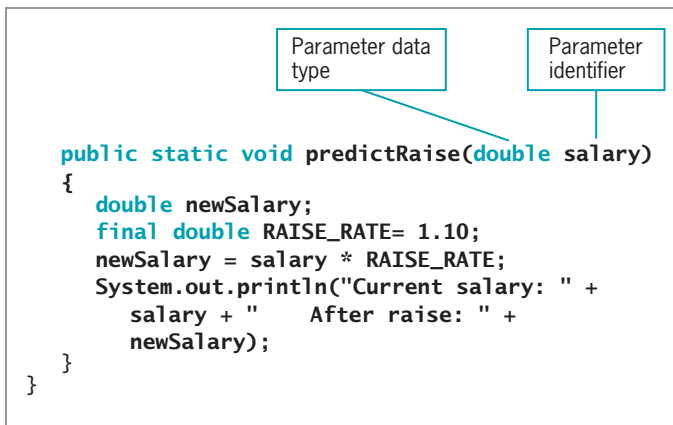
- The parameter type
- A local name for the parameter

For example, the declaration for a public method named `predictRaise()` that accepts a person's annual salary and computes the value of a 10 percent raise could be written as follows:

```
public static void predictRaise(double salary)
```

You can think of the parentheses in a method declaration as a funnel into the method—parameters listed there contain data that is “dropped into” the method. A parameter accepted by a method can be any data type, including the primitive types, such as `int`, `double`, and `char`; it also can be a class type such as `String` or `PrintStream`.

In the method header for `predictRaise()`, the parameter `double salary` within the parentheses indicates that the method will receive a value of type `double`, and that within the method, the passed value will be known as `salary`. Figure 3-13 shows a complete method.



```
public static void predictRaise(double salary)
{
    double newSalary;
    final double RAISE_RATE= 1.10;
    newSalary = salary * RAISE_RATE;
    System.out.println("Current salary: " +
        salary + " After raise: " +
        newSalary);
}
```

Figure 3-13 The `predictRaise()` method

The `predictRaise()` method is a `void` method because it does not need to return a value to any other method that uses it—its only function is to receive the `salary` value, multiply it by the `RAISE_RATE` constant (1.10, which results in a 10 percent salary increase), and then display the result.

You can call the `predictRaise()` method by using either a constant value or a variable as an argument. Thus, both `predictRaise(472.25);` and `predictRaise(mySalary);` invoke the `predictRaise()` method correctly, assuming that `mySalary` is declared as a `double` variable

and is assigned an appropriate value in the calling method. You can call the `predictRaise()` method any number of times, with a different constant or variable argument each time. Each of these arguments becomes known as `salary` within the method. The identifier `salary` represents a variable that holds a copy of the value of any `double` value passed into the method.

It is interesting to note that if the value used as an argument in the method call to `predictRaise()` is a variable, it might possess the same identifier as `salary` or a different one, such as `startingWage`. For example, the code in Figure 3-14 shows three calls to the `predictRaise()` method, and Figure 3-15 shows the output. One call uses a constant, 400.00. The other two use variables—one with the same name as `salary` and the other with a different name, `startingWage`. The identifier `salary` in the `main()` method refers to a different memory location than the one in the `predictRaise()` method. The parameter `salary` is simply a placeholder while it is being used within the `predictRaise()` method, no matter what name its value “goes by” in the calling method. The parameter `salary` is a **local variable** to the `predictRaise()` method; that is, it is known only within the boundaries of the method. The variable and constant declared within the method are also local to the method.

```
public class DemoRaise
{
    public static void main(String[] args)
    {
        double salary = 200.00;
        double startingWage = 800.00;
        System.out.println("Demonstrating some raises");
        predictRaise(400.00);
        predictRaise(salary);
        predictRaise(startingWage);
    }

    public static void predictRaise(double salary)
    {
        double newSalary;
        final double RAISE_RATE= 1.10;
        newSalary = salary * RAISE_RATE;
        System.out.println("Current salary: " +
            salary + " After raise: " +
            newSalary);
    }
}
```

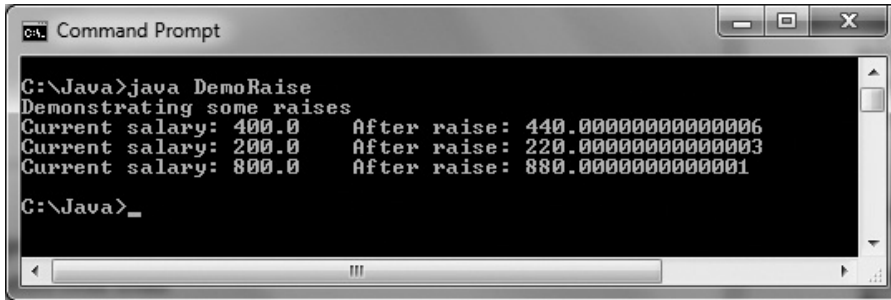
The `predictRaise()` method is called three times using three different arguments.

The parameter `salary` receives a copy of the value in each argument that is passed.

Figure 3-14 The `DemoRaise` class with a `main()` method that uses the `predictRaise()` method three times



Recall that the `final` modifier makes `RAISE_RATE` constant. Because `salary` is not altered within the `predictRaise()` method in Figure 3-14, you could also make it constant by declaring the method header as `public static void predictRaise(final double salary)`. There would be no difference in the program’s execution, but declaring a parameter as `final` means it cannot be altered within the method. Someone reading your program would be able to see that the parameter is not intended to change.



```
ca. Command Prompt
C:\Java>java DemoRaise
Demonstrating some raises
Current salary: 400.0   After raise: 440.00000000000006
Current salary: 200.0   After raise: 220.00000000000003
Current salary: 800.0   After raise: 880.00000000000001
C:\Java>_
```

Figure 3-15 Output of the DemoRaise application

Each time the `predictRaise()` method in Figure 3-14 executes, a `salary` variable is redeclared—that is, a new memory location large enough to hold a `double` is set up and named `salary`. Within the `predictRaise()` method, `salary` holds a copy of whatever value is passed into the method by the `main()` method. When the `predictRaise()` method ends at the closing curly brace, the local `salary` variable ceases to exist. That is, if you change the value of `salary` after you have used it in the calculation within `predictRaise()`, it affects nothing else. The memory location that holds `salary` is released at the end of the method, and any changes to its value within the method do not affect any value in the calling method. In particular, don't think there would be any change in the variable named `salary` in the `main()` method; that variable, even though it has the same name as the locally declared parameter in the method, is a different variable with its own memory address and is totally different from the one in the `predictRaise()` method. When a variable ceases to exist at the end of a method, programmers say the variable “goes out of scope.” A variable's scope is the part of a program in which a variable exists and can be accessed using its unqualified name. The next chapter discusses scope in greater detail.

Creating a Method That Requires Multiple Parameters

A method can require more than one parameter. You can pass multiple arguments to a method by listing the arguments within the call to the method and separating them with commas. For example, rather than creating a `predictRaise()` method that adds a 10 percent raise to every person's salary, you might prefer to create a method to which you can pass two values—the salary to be raised as well as a percentage figure by which to raise it. Figure 3-16 shows a method that uses two such parameters.

```
public static void predictRaiseUsingRate(double salary, double rate)
{
    double newAmount;
    newAmount = salary * (1 + rate);
    System.out.println("With raise, new salary is " + newAmount);
}
```

Figure 3-16 The `predictRaiseUsingRate()` method that accepts two parameters

In Figure 3-16, two parameters (`double salary` and `double rate`) appear within the parentheses in the method header. A comma separates each parameter, and each parameter requires its own declared type (in this case, both are `double`) as well as its own identifier. Note that a declaration for a method that receives two or more parameters must list the type for each parameter separately, even if the parameters have the same type. When values are passed to the method in a statement such as the following, the first value passed is referenced as `salary` within the method, and the second value passed is referenced as `rate`:

```
predictRaiseUsingRate(mySalary, promisedRate);
```

Therefore, arguments passed to the method must be passed in the correct order. The call `predictRaiseUsingRate(200.00, 0.10);` results in output representing a 10 percent raise based on a \$200.00 salary amount (or \$220.00), but `predictRaiseUsingRate(0.10, 200.00);` results in output representing a 20,000 percent raise based on a salary of 10 cents (or \$20.10). If two method parameters are the same type—for example, two `doubles`—passing arguments to a method in the wrong order results in a logical error; that is, the program does compile and execute, but it probably produces incorrect results. If a method expects parameters of diverse types, passing arguments in the wrong order constitutes a syntax error, and the program does not compile.

You can write a method so that it takes any number of parameters in any order. However, when you call a method, the arguments you send to a method must match in order—both in number and in type—the parameters listed in the method declaration. A method's **signature** is the combination of the method name and the number, types, and order of arguments. A method call must match the called method's signature.

Thus, a method to compute an automobile salesperson's commission amount might require arguments such as an integer dollar value of a car sold, a `double` percentage commission rate, and a character code for the vehicle type. The correct method executes only when three arguments of the correct types are sent in the correct order. Figure 3-17 shows a class containing a three-parameter method and a `main()` method that calls it twice, once using variable arguments and again using constant arguments. Figure 3-18 shows the output of the application.

```
public class ComputeCommission
{
    public static void main(String[] args)
    {
        char vType = 'S';
        int value = 23000;
        double commRate = 0.08;
        computeCommission(value, commRate, vType);
        computeCommission(40000, 0.10, 'L');
    }
}
```

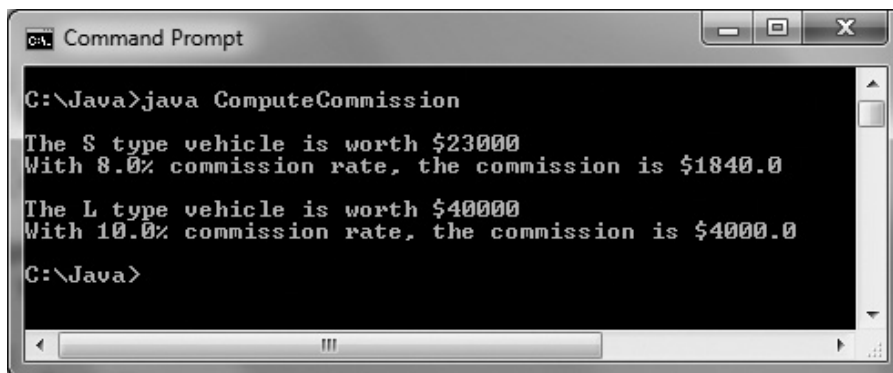
Figure 3-17 The `ComputeCommission` class (continues)

(continued)

```
public static void computeCommission(int value,
    double rate, char vehicle)
{
    double commission;
    commission = value * rate;
    System.out.println("\nThe " + vehicle +
        " type vehicle is worth $" + value);
    System.out.println("With " + (rate * 100) +
        "% commission rate, the commission is $" +
        commission);
}
```

132

Figure 3-17 The ComputeCommission class



```
C:\Java>java ComputeCommission
The S type vehicle is worth $23000
With 8.0% commission rate, the commission is $1840.0

The L type vehicle is worth $40000
With 10.0% commission rate, the commission is $4000.0
C:\Java>
```

Figure 3-18 Output of the ComputeCommission application



The arguments in a method call are often referred to as **actual parameters**. The variables in the method declaration that accept the values from the actual parameters are **formal parameters**.



When you look at Java applications, you might see methods that appear to be callable in multiple ways. For example, you can use `System.out.println()` with no arguments to display a blank line, or with a `String` argument to display the `String`. You can use the method with different argument lists only because multiple versions of the method have been written, each taking a specific set of arguments. The ability to execute different method implementations by altering the argument used with the method name is known as *method overloading*, a concept you will learn about in the next chapter.

TWO TRUTHS & A LIE

Adding Parameters to Methods

1. A class can contain any number of methods, and each method can be called any number of times.
2. Arguments are used in method calls; they are passed to parameters in method headers.
3. A method header always contains a return type, an identifier, and a parameter list within parentheses.

The false statement is #3. A method header always contains a return type, an identifier, and parentheses, but the parameter list might be empty.

Creating Methods That Return Values

A method ends when any of the following events takes place:

- The method completes all of its statements. You have seen methods like this in the last section.
- The method throws an exception. Exceptions are errors; you will learn about them in the chapter *Exception Handling*.
- The method reaches a return statement. A **return statement** causes a method to end and the program's logic to return to the calling method. Also, a return statement frequently sends a value back to the calling method.

The return type for a method can be any type used in Java, which includes the primitive types `int`, `double`, `char`, and so on, as well as class types (including class types you create). Of course, a method can also return nothing, in which case the return type is `void`.

A method's return type is known more succinctly as a **method's type**. For example, the declaration for the `nameAndAddress()` method shown earlier in Figure 3-4 is written as follows:

```
public static void nameAndAddress()
```

This method returns no value, so it is type `void`.

A method that prompts a user for an age and returns the age to the calling method might be declared as:

```
public static int getAge()
```

The method returns an `int`, so it is type `int`.

As another example, a method that returns `true` or `false`, depending on whether an employee worked overtime hours, might be declared as:

```
public static boolean workedOvertime()
```

This method returns a Boolean value, so it is type `boolean`.

The `predictRaise()` method shown earlier produces output but does not return any value, so its return type is `void`. If you want to create a method to return the new, calculated salary value rather than display it, the header would be written as follows:

```
public static double predictRaise(double salary)
```

Figure 3-19 shows this method.

```
public static double predictRaise(double salary)
{
    double newAmount;
    final double RAISE = 1.10;
    newAmount = salary * RAISE;
    return newAmount;
}
```

Figure 3-19 The `predictRaise()` method returning a `double`

Notice the shaded return type `double` that precedes the method name in the `predictRaise()` method header in Figure 3-19. Also notice the shaded declaration of `newAmount`. This `double` variable is returned in the last shaded statement in the method. The `return` statement causes a value to be sent from a called method back to the calling method. In this case, the value stored in `newAmount` is sent back to any method that calls the `predictRaise()` method. A method's declared return type must match the type of the value used in the `return` statement; if it does not, the class does not compile. A method can return, at most, one value. The value can be a primitive data type, such as `int`, `char`, or `double`, or it can be a type that represents a class. (You will learn more about class types later in this chapter.) The returned value can even be a named or unnamed constant, as long as it is the correct data type or can be automatically converted to the correct data type. For example, a method with a `double` return type could include the statement `return 1;` or `return 1.0;`.



All methods except `void` methods require a `return` statement that returns a value of the appropriate type. You can place a `return` statement in a `void` method that is simply the word `return` followed by a semicolon. However, most Java programmers do not include a `return` statement in a method when nothing is returned.

You cannot place any statements after a method's `return` statement. Such statements are **unreachable statements** because the logical flow leaves the method at the `return` statement. An unreachable statement can never execute, and it causes a compiler error. Unreachable statements are also called **dead code**.



A method can contain multiple `return` clauses if they are embedded in a decision, although this practice is not recommended because it can lead to errors that are difficult to detect. However, no other statements can be placed after the last `return` clause in a method. You will learn about decision making in the chapter *Making Decisions*.

If a method returns a value, then when you call the method, you normally use the returned value, although you are not required to do so. For example, when you invoke the `predictRaise()` method, you might want to assign the returned value (also called the method's value) to a `double` variable named `myNewSalary`, as in the following statement:

```
myNewSalary = predictRaise(mySalary);
```

The `predictRaise()` method returns a `double`, so it is appropriate to assign the method's returned value to a `double` variable.

Alternatively, you can choose to use a method's returned value directly, without storing it in any variable. When you use a method's value, you use it in the same way you would use any variable of the same type. For example, you can display a return value in a statement such as the following:

```
System.out.println("New salary is " + predictRaise(mySalary));
```

In the preceding statement, the call to the `predictRaise()` method is made from within the `println()` method call. Because `predictRaise()` returns a `double`, you can use the method call in the same way that you would use any simple `double` value. As another example, you can perform arithmetic with a method's return value, as in the following statement:

```
double spendingMoney = predictRaise(mySalary) - expenses;
```

Chaining Method Calls

Any method might call any number of other methods. For example, a `main()` method might call a `predictRaise()` method, and the `predictRaise()` method might call a `calculateBonus()` method, as shown in the shaded statement in Figure 3-20.

```
public static double predictRaise(double salary)
{
    double newAmount;
    double bonusAmount;
    final double RAISE = 1.10;
    newAmount = salary * RAISE;
    bonusAmount = calculateBonus(newAmount);
    newAmount = newAmount + bonusAmount;
    return newAmount;
}
```

Figure 3-20 The `predictRaise()` method calling the `calculateBonus()` method

Looking at the call to the `calculateBonus()` method from the `predictRaise()` method, you do not know how `calculateBonus()` works. You only know that the `calculateBonus()`

method accepts a `double` as a parameter (because `newAmount` is passed into it) and that it must return either a `double` or a type that can automatically be promoted to a `double` (because the result is stored in `bonusAmount`). In other words, the method acts as a black box.

As examples, the `calculateBonus()` method might look like either of the versions shown in Figure 3-21. The first version simply adds a \$50.00 bonus to a `salary` parameter. The second version calls another method named `trickyCalculation()` that returns a value added to `salary`. When you read the `calculateBonus()` method, you don't know what happens in the `trickyCalculation()` method, and when you read the `predictRaise()` method that calls `calculateBonus()`, you don't even necessarily know that `trickyCalculation()` is called.

```
public static double calculateBonus(double salary)
{
    final double BONUS_AMT = 50.00;
    salary = salary + BONUS_AMT;
    return salary;
}

public static double calculateBonus(double salary)
{
    salary = salary + trickyCalculation();
    return salary;
}
```

Figure 3-21 Two possible versions of the `calculateBonus()` method



Watch the video *Methods and Parameters*.

TWO TRUTHS & A LIE

Creating Methods That Return Values

1. The return type for a method can be any type used in Java, including `int`, `double`, and `void`.
2. A method's declared return type must match the type of the value used in the parameter list.
3. You cannot place a method within another method, but you can call a method from within another method.

The false statement is #2. A method's declared return type must match the type of the value used in the return statement.



You Do It

Creating static Methods That Accept Arguments and Return a Value

In this section, you add a method to the `ParadiseInfo` class you started in the last “You Do It” section. The new method receives two parameters and returns a value. The purpose of the method is to accept a minimum price for the current week’s featured discount and the percentage discount, and to return the minimum amount the customer will save.

1. Open the **ParadiseInfo.java** file in your text editor, and then change the class name to **ParadiseInfo2**. Immediately save the file as **ParadiseInfo2.java**.
2. As the first line of the file, add the `import` statement that allows user input:
import java.util.Scanner;
3. Add four declarations as the first statements following the opening curly brace of the `main()` method. One holds the minimum price for which a discount will be allowed, and another holds the discount rate. The third variable is the minimum savings, which is calculated by multiplying the minimum price for a discount and the discount rate. The fourth variable is a `Scanner` object to use for keyboard input.

```
double price;  
double discount;  
double savings;  
Scanner keyboard = new Scanner(System.in);
```



Instead of importing the `Scanner` class to provide console input, you could substitute `JOptionPane` and include program statements that provide GUI input. The input process can use other techniques too, such as getting data from a storage device—you will learn about file input in the chapter *File Input and Output*. The concept of input (getting data into memory from the outside) is the same, no matter what specific technique or type of hardware device you use.

4. Following the declarations, prompt the user for the minimum discount price, and accept a value from the keyboard:

```
System.out.print("Enter cutoff price for discount >> ");  
price = keyboard.nextDouble();
```

5. Prompt the user for the discount rate, and accept it.

```
System.out.print("Enter discount rate as a whole number >> ");  
discount = keyboard.nextDouble();
```

(continues)

(continued)

6. After the call to `displayInfo()`, insert a call to `computeDiscountInfo()`. You will pass the `price` and `discount` values to the method, and the method returns the minimum that a consumer will save, which is stored in `savings`:

```
savings = computeDiscountInfo(price, discount);
```

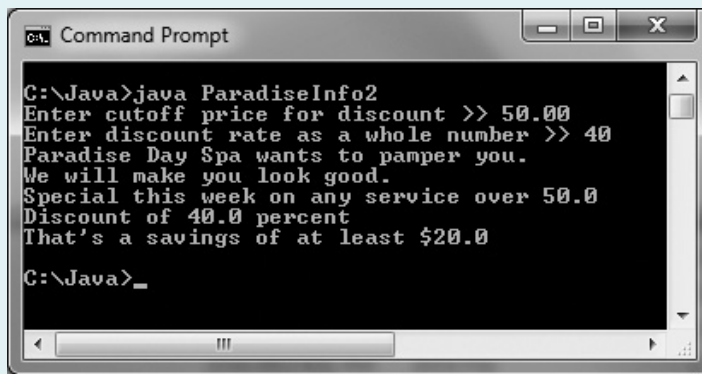
7. Just before the closing curly brace for the `main()` method, display the savings information:

```
System.out.println("Special this week on any service over " +  
    price);  
System.out.println("Discount of " + discount + " percent");  
System.out.println("That's a savings of at least $" + savings);
```

8. After the `displayInfo()` method implementation, but before the closing curly brace for the class, add the `computeDiscountInfo()` method. It accepts two `doubles` and returns a `double`.

```
public static double computeDiscountInfo(double pr, double dscnt)  
{  
    double savings;  
    savings = pr * dscnt / 100;  
    return savings;  
}
```

9. Save the file, and then compile and execute it. Figure 3-22 shows a typical execution. After the user is prompted for the cutoff price for the week's sale and the discount to be applied, the program executes the `displayInfo()` method. Then the program executes the `computeDiscountInfo()` method, which returns a value to store in the `savings` variable. Finally, the discount information is displayed.



```
C:\Java>java ParadiseInfo2  
Enter cutoff price for discount >> 50.00  
Enter discount rate as a whole number >> 40  
Paradise Day Spa wants to pamper you.  
We will make you look good.  
Special this week on any service over 50.0  
Discount of 40.0 percent  
That's a savings of at least $20.0  
C:\Java>_
```

Figure 3-22 Typical execution of the `ParadiseInfo2` program

(continues)

(continued)

Understanding That Methods Can Be Used as Black Boxes

In this chapter, you have learned that methods can be used without knowing the details of their implementation. As an example of how professional programmers use implementation hiding, you can visit the Java Web site to see the interfaces for thousands of prewritten methods that reside in the Java prewritten classes. You are not allowed to see the code inside these methods; you see only their interfaces, which is all you need to be able to use the methods.

1. Open your Web browser, go to the Java Web site, and navigate to the Java APIs for Java SE 7.
2. From the alphabetical list of classes, select **PrintStream**.
3. At the `PrintStream` page, read the descriptions of several methods for the class. Note that for each method, you can see the return type, method name, and parameter list, but you do not see the implementation for any of the existing methods.
4. Examine other classes. Again, note that the Java documentation provides you with method interfaces but not implementations. When you develop your own classes in the future, you might choose to provide your users with similar documentation and compiled classes so that they cannot see, modify, or steal the code you have worked hard to develop.

Learning About Classes and Objects

When you think in an object-oriented manner, everything is an object, and every object is a member of a class. You can think of any inanimate physical item as an object—your desk, your computer, and the building in which you live are all called objects in everyday conversation. You can also think of living things as objects—your houseplant, your pet fish, and your sister are objects. Events are also objects—the stock purchase you made, the mortgage closing you attended, and a graduation party that was held in your honor are all objects.

Everything is an object, and every object is a member of a more general class. Your desk is a member of the class that includes all desks, and your pet fish is a member of the class that contains all fish. An object-oriented programmer would say that your desk is an instance of the `Desk` class and your fish is an instance of the `Fish` class. These statements represent **is-a relationships**—that is, relationships in which the object “is a” concrete example of the class. Expressing an is-a relationship is correct only when you refer to the object and the class in the proper order. You can say, “My oak desk with the scratch on top *is a* `Desk`, and my goldfish named Moby *is a* `Fish`.” You don’t define a `Desk` by saying, “A `Desk` *is an* oak desk with a scratch on top,” or explain what a `Fish` is by saying, “A `Fish` *is a* goldfish named Moby,” because both a `Desk` and a `Fish` are much more general. The difference between a class and

an object parallels the difference between abstract and concrete. An object is an **instantiation** of a class, or one tangible example of a class. Your goldfish, my guppy, and the zoo's shark each constitute one instantiation of the `Fish` class.



Programmers also use the phrase “is-a” when talking about inheritance relationships. You will learn more about inheritance in the chapters *Introduction to Inheritance* and *Advanced Inheritance Concepts*.

The concept of a class is useful because of its reusability. Objects gain their attributes from their classes, and all objects have predictable attributes because they are members of certain classes. For example, if you are invited to a graduation party, you automatically know many things about the object (the party). You assume there will be a starting time, a certain number of guests, and some quantity of food. You understand what a party entails because of your previous knowledge of the `Party` class of which all parties are members. You don't know the number of guests or what food will be served at this particular party, but you understand that because all parties have guests and refreshments, this one must too. Because you understand the general characteristics of a `Party`, you anticipate different behaviors than if you plan to attend a `TheaterPerformance` object or a `DentalAppointment` object.

In addition to their attributes, objects have methods associated with them, and every object that is an instance of a class is assumed to possess the same methods. For example, for all `Party` objects, at some point you must set the date and time. In a program, you might name these methods `setDate()` and `setTime()`. Party guests need to know the date and time and might use methods named `getDate()` and `getTime()` to find out the date and time of any `Party` object. Method names that begin with *get* and *set* are very typical. You will learn more about *get* and *set* methods in the next section.

Your graduation party, then, might have the identifier `myGraduationParty`. As a member of the `Party` class, `myGraduationParty`, like all `Party` objects, might have data methods `setDate()` and `setTime()`. When you use them, the `setDate()` and `setTime()` methods require arguments, or information passed to them. For example, statements such as `myGraduationParty.setDate("May 12")` and `myGraduationParty.setTime("6 P.M.")` invoke methods that are available for the `myGraduationParty` object. When you use an object and its methods, think of being able to send a message to the object to direct it to accomplish some task—you can tell the `Party` object named `myGraduationParty` to set the date and time you request. Even though `yourAnniversaryParty` is also a member of the `Party` class, and even though it also has access to `setDate()` and `setTime()` methods, the arguments you send to `yourAnniversaryParty` methods will be different from those you send to `myGraduationParty` methods. Within any object-oriented program, you are continuously making requests to objects' methods and often including arguments as part of those requests.

In addition, some methods used in an application must return a message or value. If one of your party guests uses the `getDate()` method, the guest hopes that the method will respond with the desired information. Similarly, within object-oriented programs, methods are often called upon to return a piece of information to the source of the request. For example, a method within a `Payroll` class that calculates federal withholding tax might return a tax

amount in dollars and cents, and a method within an `Inventory` class might return `true` or `false`, depending on the method's determination of whether an item is at the reorder point.

With object-oriented programming, sometimes you create classes so that you can instantiate objects from them, and other times you create classes to run as applications. Application classes frequently instantiate objects that use the objects of other classes (and their data and methods). Sometimes you write classes that do both. The same programmer does not need to write every class he or she uses. Often, you will write programs that use classes created by others; similarly, you might create a class that others will use to instantiate objects within their own applications. You can call an application or class that instantiates objects of another prewritten class a **class client** or **class user**.



The `System` class that you have used to produce output in the Command Prompt window provides an example of using a class that was written by someone else. You did not have to create it or its `println()` method; both were created for you by Java's creators.



You can identify a class that is an application because it contains a `public static void main()` method. The `main()` method is the starting point for any application. You will write and use many classes that do not contain a `main()` method—these classes can be used by other classes that are applications or applets. (You will learn about applets in the chapter *Applets, Images, and Sound*.)



A Java application can contain only one method with the header `public static void main(String[] args)`. If you write a class that imports another class, and both classes have a `public main()` method, your application will not compile.

So far, you've learned that object-oriented programming involves objects that send messages to other objects requesting they perform tasks, and that every object belongs to a class. Understanding classes and how objects are instantiated from them is the heart of object-oriented thinking.

TWO TRUTHS & A LIE

Learning About Classes and Objects

1. A class is an instantiation of many objects.
2. Objects gain their attributes and methods from their classes.
3. An application or class that instantiates objects of another prewritten class is a class client.

The false statement is #1. An object is one instantiation of a class.

Creating a Class

When you create a class, you must assign a name to the class, and you must determine what data and methods will be part of the class. Suppose you decide to create a class named `Employee`. One instance variable of `Employee` might be an employee number, and two necessary methods might be a method to set (or provide a value for) the employee number and another method to get (or retrieve) that employee number. To begin, you create a class header with three parts:

- An optional access specifier
- The keyword `class`
- Any legal identifier you choose for the name of your class

For example, a header for a class that represents an employee might be:

```
public class Employee
```

The most liberal form of access is `public`. The keyword `public` is a class modifier. Classes that are `public` are accessible by all objects. Public classes also can be **extended**, or used as a basis for any other class. Making access `public` means that if you develop a good `Employee` class, and someday you want to develop two classes that are more specific, `SalariedEmployee` and `HourlyEmployee`, then you do not have to start from scratch. Each new class can become an extension of the original `Employee` class, inheriting its data and methods. Although other specifiers exist, you will use the `public` specifier for most of your classes.



You will learn about extended classes in the chapter *Introduction to Inheritance*.

After writing the class header `public class Employee`, you write the body of the `Employee` class between a set of curly braces. The body contains the data and methods for the class. The data components of a class are often referred to as **data fields** to help distinguish them from other variables you might use. Figure 3-23 shows an `Employee` class that contains one data field named `empNum`. Data fields are variables you declare within a class but outside of any method.

```
public class Employee
{
    private int empNum;
}
```

Figure 3-23 The `Employee` class with one field

In Figure 3-23, the data field `empNum` is not preceded by the keyword `static`. If the keyword `static` had been inserted there, only one `empNum` value would be shared by all `Employee` objects that are eventually instantiated. Because the `empNum` field in Figure 3-23 is not preceded by `static`, when you eventually create, or instantiate, objects from the class, each `Employee` can have its own unique `empNum`. Each object gets its own copy of each nonstatic data field. A nonstatic field like `empNum` is an **instance variable** for the class.

You have already learned that the access specifier for most Java methods is `public`. However, most fields, like `empNum` in the `Employee` class, are `private`, which provides the highest level of security. Assigning **private access** to a field means that no other classes can access the field's values, and only methods of the same class are allowed to set, get, or otherwise use private variables. The principle used in creating private access is sometimes called **information hiding** and is an important component of object-oriented programs. A class's private data can be changed or manipulated only by a class's own methods and not by methods that belong to other classes. In contrast to fields, most class methods are `public`, not `private`. The resulting private data/public method arrangement provides a means for you to control outside access to your data—only a class's nonprivate methods can be used to access a class's private data. The situation is similar to hiring a public receptionist to sit in front of your private office and control which messages you receive (perhaps deflecting trivial or hostile ones) and which messages you send (perhaps checking your spelling, grammar, and any legal implications). The way in which the nonprivate methods are written controls how you use the private data.



The first release of Java (1.0) supported five access levels—the four listed previously plus `private protected`. The `private protected` access level is not supported in versions of Java higher than 1.0; you should not use it in your Java programs.

In summary, a class's data fields are most often private and not static. Data fields are most frequently made public when they are both `static` and `final`—that is, when a class contains a nonchanging value that you want to use without being required to create an object. For example, the Java `Math` class contains a `public` field called `PI` that you can use without instantiating a `Math` object. You will learn about the `Math` class in the next chapter.

TWO TRUTHS & A LIE

Creating a Class

1. A class header contains an optional access specifier, the keyword `class`, and an identifier.
2. When you instantiate objects, each has its own copy of each `static` data field in the class.
3. Most fields in a class are `private`, and most methods are `public`.

The false statement is #2. When you instantiate objects, each has its own copy of each nonstatic data field in the class.

Creating Instance Methods in a Class

Besides data, classes contain methods. For example, one method you need for an `Employee` class that contains an `empNum` is the method to retrieve (or return) any `Employee`'s `empNum` for

use by another class. A reasonable name for this method is `getEmpNum()`, and its declaration is `public int getEmpNum()` because it will have `public` access, return an integer (the employee number), and possess the identifier `getEmpNum()`.

Similarly, you need a method with which to set the `empNum` field. A reasonable name for this method is `setEmpNum()`, and its declaration is `public void setEmpNum(int emp)` because it will have `public` access, return nothing, possess the identifier `setEmpNum()`, and require a parameter that represents the employee's ID number, which is type `int`.

Methods that set or change field values are called **mutator methods**; methods that retrieve values are called **accessor methods**. In Java, mutator methods conventionally start with the prefix *set*, and accessor methods conventionally start with the prefix *get*. Using these three-letter prefixes with your method names is not required, but it is conventional. Figure 3-24 shows the `get` and `set` methods for the `empNum` field for the `Employee` class.

```
public void setEmpNum(int emp)
{
    empNum = emp;
}

public int getEmpNum()
{
    return empNum;
}
```

Figure 3-24 The `setEmpNum()` and `getEmpNum()` methods

Notice that, unlike the methods you created earlier in this chapter, the `getEmpNum()` and `setEmpNum()` methods do not employ the `static` modifier. The keyword `static` is used for classwide methods, but not for methods that “belong” to objects. If you are creating a program with a `main()` method that you will execute to perform some task, many of your methods will be `static` so you can call them from within `main()` without creating objects. However, if you are creating a class from which objects will be instantiated, most methods will probably be `nonstatic` because you will associate the methods with individual objects. For example, the `getEmpNum()` method must be `nonstatic` because it returns a different `empNum` value for every `Employee` object you ever create. **Nonstatic methods**, those methods used with object instantiations, are called **instance methods**. You *can* use either a `static` or `nonstatic` method with an object, but only `nonstatic` methods behave uniquely for each object. You cannot use a `nonstatic` method without an object.

Understanding when to declare fields and methods as `static` and `nonstatic` is a challenge for new programmers. To help you determine whether a data field should be `static` or not, you can ask yourself how many times it occurs. If it occurs once per class, it is `static`, but if it occurs once per object, it is not `static`. Table 3-1 provides a summary.

Static	Nonstatic
<p>In Java, <code>static</code> is a keyword. It also can be used as an adjective.</p>	<p>There is no keyword for nonstatic items. When you do not explicitly declare a field or method to be static, then it is nonstatic by default.</p>
<p>Static fields in a class are called class fields.</p>	<p>Nonstatic fields in a class are called instance variables.</p>
<p>Static methods in a class are called class methods.</p>	<p>Nonstatic methods in a class are called instance methods.</p>
<p>When you use a static field or method, you do not need to use an object; for example: <code>JOptionPane.showDialog();</code></p>	<p>When you use a nonstatic field or method, you must use an object; for example: <code>System.out.println();</code></p>
<p>When you create a class with a static field and instantiate 100 objects, only one copy of that field exists in memory.</p>	<p>When you create a class with a nonstatic field and instantiate 100 objects, then 100 copies of that field exist in memory.</p>
<p>When you create a static method in a class and instantiate 100 objects, only one copy of the method exists in memory and the method does not receive a <code>this</code> reference.</p>	<p>When you create a nonstatic method in a class and instantiate 100 objects, only one copy of the method exists in memory, but the method receives a <code>this</code> reference that contains the address of the object currently using it.</p>
<p>Static class variables are not instance variables. The system allocates memory to hold class variables once per class, no matter how many instances of the class you instantiate. The system allocates memory for class variables the first time it encounters a class, and every instance of a class shares the same copy of any static class variables.</p>	<p>Instance fields and methods are nonstatic. The system allocates a separate memory location for each nonstatic field in each instance.</p>

Table 3-1 Comparison of static and nonstatic



Table 3-1 mentions the `this` reference. You will learn about the `this` reference in the next chapter.

Figure 3-25 also provides a summary of how `public`, `private`, `static`, and nonstatic class members can be used by another class. The figure shows a class named `MyClass` with four methods that are `public static`, `private static`, `public nonstatic`, and `private nonstatic`. The figure also shows a `TestClass` that instantiates a `MyClass` object. The `TestClass` contains eight method calls. The three valid calls are all to `public` methods. The call to the

nonstatic method uses an object, and the two calls to the `static` method can use an object or not. The rest of the `TestClass` code after the comment is invalid. Private methods cannot be called from outside the class, and nonstatic methods require an object.

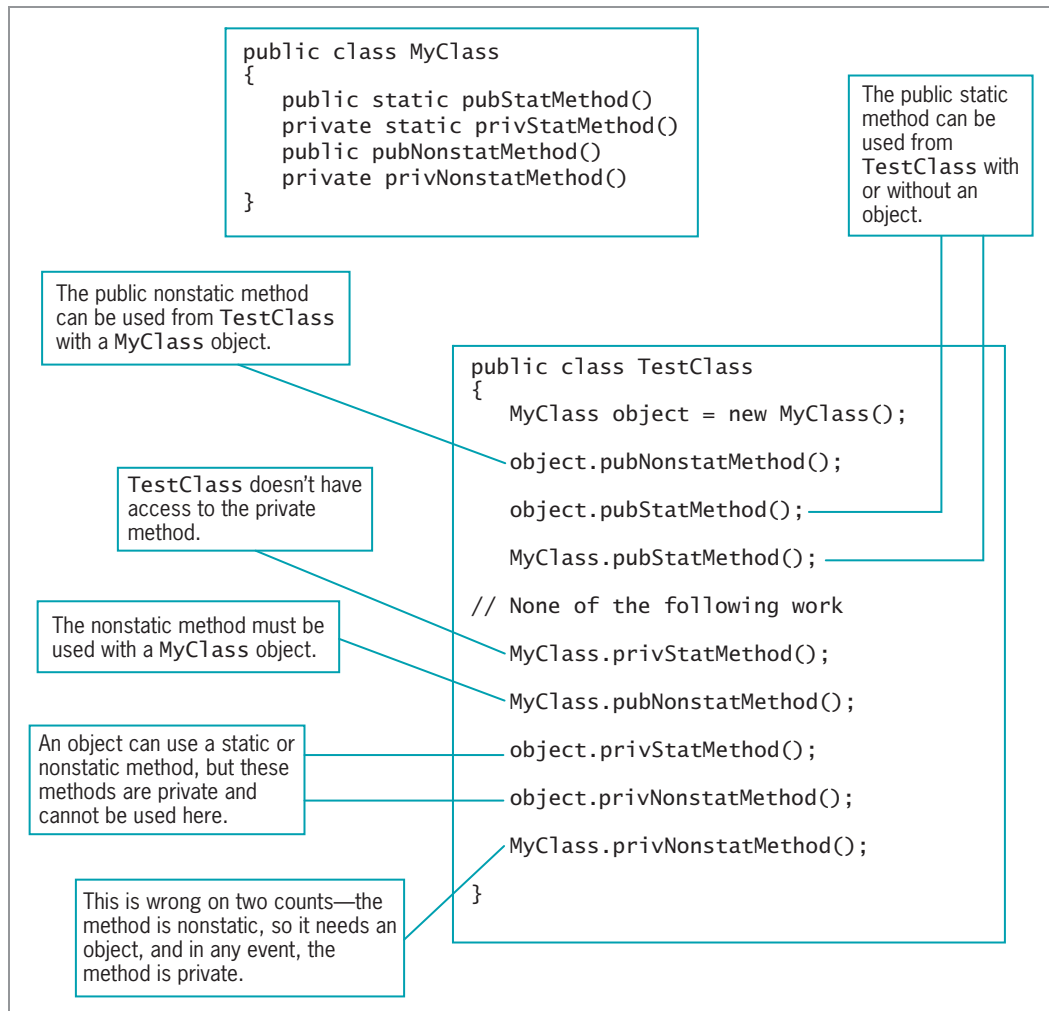


Figure 3-25 Summary of legal and illegal method calls based on combinations of method modifiers

Figure 3-26 shows the complete `Employee` class containing one private data field and two public methods, all of which are nonstatic. This class becomes the model for a new data type named `Employee`; when `Employee` objects eventually are created, each will have its own `empNum` field, and each will have access to two methods—one that provides a value for its `empNum` field and another that retrieves the value stored there.

```
public class Employee
{
    private int empNum;
    public int getEmpNum()
    {
        return empNum;
    }
    public void setEmpNum(int emp)
    {
        empNum = emp;
    }
}
```

Figure 3-26 The Employee class with one field and two methods

When you create a class like Employee, you can compile it, which will identify typographical errors. However, you cannot execute the class because it does not contain a main() method. A class like Employee is intended to be used as a data type for objects within other applications, as you will see in the next section.

Organizing Classes

Most classes that you create have multiple data fields and methods. For example, in addition to requiring an employee number, an Employee needs a last name, a first name, and a salary, as well as methods to set and get those fields. Figure 3-27 shows how you could code the data fields for Employee.

```
public class Employee
{
    private int empNum;
    private String empLastName;
    private String empFirstName;
    private double empSalary;
    // Methods will go here
}
```

Figure 3-27 An Employee class with several data fields

Although there is no requirement to do so, most programmers place data fields in some logical order at the beginning of a class. For example, empNum is most likely used as a unique identifier for each employee (what database users often call a **primary key**), so it makes sense to list the employee number first in the class. An employee's last name and first name "go together," so it makes sense to store these two Employee components adjacently. Despite these commonsense rules, you have a lot of flexibility in how you position your data fields within any class.



A unique identifier is one that should have no duplicates within an application. For example, an organization might have many employees with the last name Johnson or a weekly salary of \$400.00, but there is only one employee with employee number 128.

Because there are two `String` components in the current `Employee` class, they might be declared within the same statement, such as the following:

```
private String empLastName, empFirstName;
```

However, it is usually easier to identify each `Employee` field at a glance if the fields are listed vertically.

You can place a class's data fields and methods in any order within a class. For example, you could place all the methods first, followed by all the data fields, or you could organize the class so that several data fields are followed by methods that use them, and then several more data fields are followed by the methods that use them. This book follows the convention of placing all data fields first so that you can see their names and data types before reading the methods that use them.

Even if the only methods in the `Employee` class started in Figure 3-27 include one `set` method and one `get` method for each instance variable, eight methods are required. When you consider an employee record for most organizations, you will realize that many more fields are often required (such as address, phone number, hire date, number of dependents, and so on), as well as many more methods. Finding your way through the list can become a formidable task. For ease in locating class methods, many programmers store them in alphabetical order. Other programmers arrange values in pairs of `get` and `set` methods, an order that also results in functional groupings. Figure 3-28 shows how the complete class definition for an `Employee` might appear.

```
public class Employee
{
    private int empNum;
    private String empLastName;
    private String empFirstName;
    private double empSalary;
    public int getEmpNum()
    {
        return empNum;
    }
    public void setEmpNum(int emp)
    {
        empNum = emp;
    }
    public String getEmpLastName()
    {
        return empLastName;
    }
    public void setEmpLastName(String name)
    {
        empLastName = name;
    }
}
```

Figure 3-28 The `Employee` class with several data fields and corresponding methods (*continues*)

(continued)

```

public String getEmpFirstName()
{
    return empFirstName;
}
public void setEmpFirstName(String name)
{
    empFirstName = name;
}
public double getEmpSalary()
{
    return empSalary;
}
public void setEmpSalary(double sal)
{
    empSalary = sal;
}
}

```

Figure 3-28 The Employee class with several data fields and corresponding methods

The Employee class is still not a particularly large class, and each of its methods is very short, but it is already becoming quite difficult to manage. It certainly can support some well-placed comments. For example, the purpose of the class and the programmer's name might appear in comments at the top of the file, and comments might be used to separate the data and method sections of the class. Your organization might have specific recommendations or requirements for placing comments within a class.

TWO TRUTHS & A LIE

Creating Instance Methods in a Class

1. The keyword `static` is used with classwide methods, but not for methods that “belong” to objects.
2. When you create a class from which objects will be instantiated, most methods are nonstatic because they are associated with individual objects.
3. Static methods are instance methods.

The false statement is #3. Nonstatic methods are instance methods; static methods are class methods.



You Do It

150

Creating a Class That Contains Instance Fields and Methods

Next, you create a class to store information about event services offered at Paradise Day Spa.

1. Open a new document in your text editor, and type the following class header and the curly braces to surround the class body:

```
public class SpaService  
{  
}
```

2. Between the curly braces for the class, insert two private data fields that will hold data about a spa service:

```
private String serviceDescription;  
private double price;
```

3. Within the class's curly braces and after the field declarations, enter the following two methods that set the field values. The `setServiceDescription()` method accepts a `String` parameter and assigns it to the `serviceDescription` field for each object that eventually will be instantiated. Similarly, the `setPrice()` method accepts a `double` parameter and assigns it to the `price` field. Note that neither of these methods is `static`.

```
public void setServiceDescription(String service)  
{  
    serviceDescription = service;  
}  
public void setPrice(double pr)  
{  
    price = pr;  
}
```

4. Next, add two methods that retrieve the field values as follows:

```
public String getServiceDescription()  
{  
    return serviceDescription;  
}  
public double getPrice()  
{  
    return price;  
}
```

5. Save the file as **SpaService.java**, compile it, and then correct any syntax errors. Remember, you cannot run this file as a program because it does not contain a `public static main()` method. After you read the next section, you will use this class to create objects.

Declaring Objects and Using Their Methods

Declaring a class does not create any actual objects. A class is just an abstract description of what an object will be like if any objects are ever actually instantiated. Just as you might understand all the characteristics of an item you intend to manufacture long before the first item rolls off the assembly line, you can create a class with fields and methods long before you instantiate any objects that are members of that class.

A two-step process creates an object that is an instance of a class. First, you supply a type and an identifier—just as when you declare any variable—and then you allocate computer memory for that object. For example, you might declare an integer as `int someValue`; and you might declare an `Employee` as follows:

```
Employee someEmployee;
```

In this statement, `someEmployee` stands for any legal identifier you choose to represent an `Employee`.

When you declare an integer as `int someValue`;, you notify the compiler that an integer named `someValue` will exist, and you reserve computer memory for it at the same time. When you declare the `someEmployee` instance of the `Employee` class, you are notifying the compiler that you will use the identifier `someEmployee`. However, you are not yet setting aside computer memory in which the `Employee` named `someEmployee` might be stored—that is done automatically only for primitive type variables. To allocate the needed memory for an object, you must use the **new operator**. Two statements that actually set aside enough memory to hold an `Employee` are as follows:

```
Employee someEmployee;  
someEmployee = new Employee();
```



You first learned about the `new` operator when you created a `Scanner` object in Chapter 2.

Instead of using two statements, you can declare and reserve memory for `someEmployee` in one statement, as in the following:

```
Employee someEmployee = new Employee();
```

In this statement, `Employee` is the object's type (as well as its class), and `someEmployee` is the name of the object. In this statement, `someEmployee` becomes a **reference to the object**—the name for a memory address where the object is held. Every object name is also a reference—that is, a computer memory location. In Chapter 2, you learned that a class like `Employee` is a *reference type*.

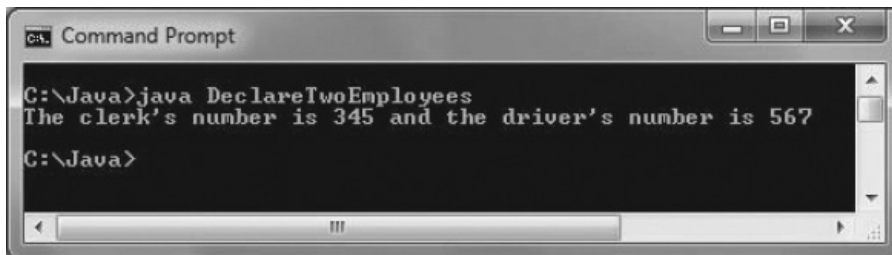
The equal sign is the assignment operator, so a value is being assigned to `someEmployee` in the declaration. The `new` operator is allocating a new, unused portion of computer memory for `someEmployee`. The value that the statement is assigning to `someEmployee` is a memory address at which `someEmployee` is to be located. You do not need to be concerned with what the actual memory address is—when you refer to `someEmployee`, the compiler locates it at the appropriate address for you.

The final portion of the statement after the `new` operator, `Employee()`, with its parentheses, looks suspiciously like a method name. In fact, it is the name of a method that constructs an `Employee` object. The `Employee()` method is a **constructor**, a special type of method that creates and initializes objects. You can write your own constructor for a class, and you will learn how later in this chapter. However, when you don't write a constructor for a class, Java writes one for you. Whether you write your own constructor or use the one automatically created by Java, the name of the constructor is always the same as the name of the class whose objects it constructs.

After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call. For example, Figure 3-29 shows an application that instantiates two `Employee` objects. The two objects, `clerk` and `driver`, each use the `setEmpNum()` and `getEmpNum()` method one time. The `DeclareTwoEmployees` application can use these methods because they are public, and it must use them with an `Employee` object because the methods are not static. Figure 3-30 shows the output of the application.

```
public class DeclareTwoEmployees
{
    public static void main(String[] args)
    {
        Employee clerk = new Employee();
        Employee driver = new Employee();
        clerk.setEmpNum(345);
        driver.setEmpNum(567);
        System.out.println("The clerk's number is " +
            clerk.getEmpNum() + " and the driver's number is " +
            driver.getEmpNum());
    }
}
```

Figure 3-29 The `DeclareTwoEmployees` class



```
C:\Java>java DeclareTwoEmployees
The clerk's number is 345 and the driver's number is 567
C:\Java>
```

Figure 3-30 Output of the `DeclareTwoEmployees` application



The program in Figure 3-29 assumes that the `Employee.java` file is stored in the same folder as the application. If the `Employee.java` file was stored in a different folder, you would need an `import` statement at the top of the file, similar to the ones you use for the `Scanner` and `JOptionPane` classes.

Understanding Data Hiding

Within the `DeclareTwoEmployees` class, you must use the public methods `setEmpNum()` and `getEmpNum()` to be able to set and retrieve the value of the `empNum` field for each `Employee` because you cannot access the private `empNum` field directly. For example, the following statement would not be allowed:

```
c1erk.empNum = 789;
```

This statement generates the error message “`empNum` has private access in `Employee`”, meaning you cannot access `empNum` from the `DeclareTwoEmployees` class. If you made `empNum` `public` instead of `private`, a direct assignment statement would work, but you would violate an important principle of object-oriented programming—that of data hiding using encapsulation. Data fields should usually be private, and a client application should be able to access them only through the public interfaces—that is, through the class’s public methods. However, you might reasonably ask, “When I write an application, if I *can’t* set an object’s data field directly, but I *can* set it using a public method, what’s the difference? The field value is set either way!” Actually, the `setEmpNum()` method in the `Employee` class in Figure 3-26 *does* accept any integer value you send into it. However, you could rewrite the `setEmpNum()` method to prevent invalid data from being assigned to an object’s data fields. For example, perhaps your organization has rules for valid employee ID numbers—they must be no fewer than five digits, or they must start with a 9, for instance—or perhaps you calculate a check-digit that is appended to every employee ID number. The statements that enforce these requirements would be part of the `setEmpNum()` method. Checking a value for validity requires decision making. You will learn more in the chapter *Making Decisions*.



A check-digit is a number appended to a field, typically an ID number or account number. The check-digit ensures that the number is valid. For example, an organization might use five-digit employee ID numbers in which the fifth digit is calculated by dividing the first four by 7 and taking the remainder. As an example, if the first four digits of your ID number are 7235, then the fifth digit is 4, the remainder when you divide the first four digits by 7. So the five-digit ID becomes 72354. Later, if you make a mistake and enter your ID into a company application as 82354, the application would divide the first four digits, 8235, by 7. The remainder is not 4, and the ID would be found invalid.

Similarly, a `get` method might control how a value is retrieved. Perhaps you do not want clients to have access to part of an employee’s ID number, or perhaps you always want to add a company code to every ID before it is returned to the client. Even when a field has no data value requirements or restrictions, making data private and providing public `set` and `get` methods establishes a framework that makes such modifications easier in the future. You will not necessarily write `set` and `get` methods for every field in a class; there are some fields that clients will not be allowed to alter. Some fields will simply be assigned values, and some field values might be calculated from the values of others.



Watch the video *Classes and Objects*.

TWO TRUTHS & A LIE

Declaring Objects and Using Their Methods

1. When you declare an object, you give it a name and set aside enough memory for the object to be stored.
2. An object name is a reference; it holds a memory address.
3. When you don't write a constructor for a class, Java creates one for you; the name of the constructor is always the same as the name of its class.

The false statement is #1. When you declare an object, you are not yet setting aside computer memory in which the object is stored; to allocate the needed memory for an object, you must use the new operator.



You Do It

Declaring and Using Objects

In the previous “You Do It” section, you created a class named `SpaService`. Now you create an application that instantiates and uses `SpaService` objects.

1. Open a new file in your text editor, and type the `import` statement needed for an interactive program that accepts user keyboard input:

```
import java.util.Scanner;
```

2. Create the shell for a class named `CreateSpaServices`:

```
public class CreateSpaServices
{
}
```

3. Between the curly braces of the `CreateSpaServices` class, create the shell for a `main()` method for the application:

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

(continues)

(continued)

4. Within the `main()` method, declare variables to hold a service description and price that a user can enter from the keyboard:

```
String service;  
double price;
```

5. Next, declare three objects. Two are `SpaService` objects that use the class you created in the prior set of “You Do It” steps. The third object uses the built-in Java `Scanner` class. Both classes use the `new` operator to allocate memory for their objects, and both call a constructor that has the same name as the class. The difference is that the `Scanner` constructor requires an argument (`System.in`), but the `SpaService` class does not.

```
SpaService firstService = new SpaService();  
SpaService secondService = new SpaService();  
Scanner keyboard = new Scanner(System.in);
```

6. In the next statements, you prompt the user for a service, accept it from the keyboard, prompt the user for a price, and accept it from the keyboard.

```
System.out.print("Enter service >> ");  
service = keyboard.nextLine();  
System.out.print("Enter price >> ");  
price = keyboard.nextDouble();
```

7. Recall that the `setServiceDescription()` method in the `SpaService` class is nonstatic, meaning it is used with an object, and that it requires a `String` argument. Write the statement that sends the service the user entered to the `setServiceDescription()` method for the `firstService` object:

```
firstService.setServiceDescription(service);
```

8. Similarly, send the price the user entered to the `setPrice()` method for the `firstService` object. Recall that this method is nonstatic and requires a `double` argument.

```
firstService.setPrice(price);
```

9. Make a call to the `nextLine()` method to remove the Enter key that remains in the input buffer after the last numeric entry. Then repeat the prompts, and accept data for the second `SpaService` object.

```
keyboard.nextLine();  
System.out.print("Enter service >> ");  
service = keyboard.nextLine();  
System.out.print("Enter price >> ");  
price = keyboard.nextDouble();  
secondService.setServiceDescription(service);  
secondService.setPrice(price);
```

(continues)

(continued)

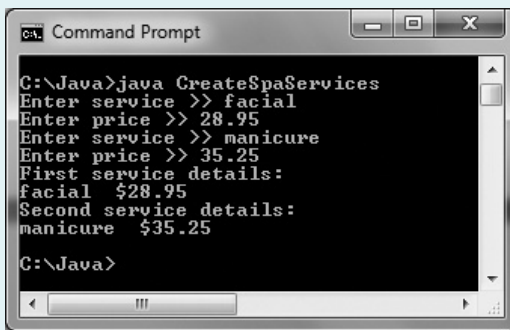
10. Display the details for the `firstService` object.

```
System.out.println("First service details:");
System.out.println(firstService.getServiceDescription() +
    " $" + firstService.getPrice());
```

11. Display the details for the `secondService` object.

```
System.out.println("Second service details:");
System.out.println(secondService.getServiceDescription() +
    " $" + secondService.getPrice());
```

12. Save the file as **CreateSpaServices.java**. Compile and execute the program. Figure 3-31 shows a typical execution. Make sure you understand how the user's entered values are assigned to and retrieved from the two `SpaService` objects.



```
CA: Command Prompt
C:\Java>java CreateSpaServices
Enter service >> facial
Enter price >> 28.95
Enter service >> manicure
Enter price >> 35.25
First service details:
facial $28.95
Second service details:
manicure $35.25
C:\Java>
```

Figure 3-31 Typical execution of the `CreateSpaServices` program

An Introduction to Using Constructors

When you create a class, such as `Employee`, and instantiate an object with a statement such as the following, you are actually calling the `Employee` class constructor that is provided by default by the Java compiler:

```
Employee chauffeur = new Employee();
```

A constructor establishes an object; a **default constructor** is one that requires no arguments. A default constructor is created automatically by the Java compiler for any class you create whenever you do not write your own constructor.

When the prewritten, default constructor for the `Employee` class is called, it establishes one `Employee` object with the identifier provided. The automatically supplied default constructor provides the following specific initial values to an object's data fields:

- Numeric fields are set to 0 (zero).
- Character fields are set to Unicode ‘\u0000’.
- Boolean fields are set to `false`.
- Fields that are object references (for example, `String` fields) are set to `null` (or empty).

If you do not want each field in an object to hold these default values, or if you want to perform additional tasks when you create an instance of a class, you can write your own constructor. Any constructor you write must have the same name as the class it constructs, and constructors cannot have a return type. You never provide a return type for a constructor—not even `void`. Normally, you declare constructors to be `public` so that other classes can instantiate objects that belong to the class. When you write a constructor for a class, you no longer have access to the automatically created version.

For example, if you want every `Employee` object to have a default starting salary of \$300.00 per week, you could write the constructor for the `Employee` class that appears in Figure 3-32. Any `Employee` object instantiated will have an `empSalary` field value equal to 300.00, and the other `Employee` data fields will contain the automatically supplied default values. Even though you might want a field to hold the default value, you still might prefer to explicitly initialize the field for clarity.

```
public Employee()  
{  
    empSalary = 300.00;  
}
```

Figure 3-32 The `Employee` class constructor



The `Employee` class constructor in Figure 3-32 takes no parameters. Therefore, it is a default constructor. You will learn about nondefault constructors that take parameters in the next chapter.

You can write any Java statement in a constructor. Although you usually have no reason to do so, you could display a message from within a constructor or perform any other task.

You can place the constructor anywhere inside the class, outside of any other method. Typically, a constructor is placed with the other methods. Often, programmers list the constructor first because it is the first method used when an object is created.

You never are required to write a constructor for a class; Java provides you with a default version if the class contains no explicit constructor.



A class can contain multiple constructors. You will learn how to overload constructors in the next chapter.



Watch the video *Constructors*.

TWO TRUTHS & A LIE

An Introduction to Using Constructors

1. In Java, you cannot write a default constructor; it must be supplied for you automatically.
2. The automatically supplied default constructor sets all numeric fields to 0, character fields to Unicode `      `, Boolean fields to `false`, and fields that are object references to `null`.
3. When you write a constructor, it must have the same name as the class it constructs, and it cannot have a return type.

The false statement is #1. A default constructor is one that takes no parameters. If you do not create a constructor for a class, Java creates a default constructor for you. However, you can create a default constructor that replaces the automatically supplied one.



You Do It

Adding a Constructor to a Class

1. Open the **SpaService.java** file that you created in a “You Do It” section earlier in this chapter.
2. After the field declarations, and before the method declarations, insert an explicit default constructor that sets `serviceDescription` to “XXX” and `price` to 0. Because numeric fields in objects are set to 0 by default, the last assignment is not really necessary. However, programmers sometimes code a statement like the one that sets `price` to 0 so that their intentions are clear to people reading their programs.

(continues)

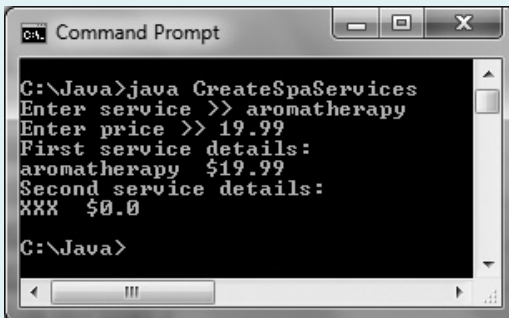
(continued)

```
public SpaService()  
{  
    serviceDescription = "XXX";  
    price = 0;  
}
```

3. Save the class and compile it.
4. Open the **CreateSpaServices.java** file. Comment out the seven statements that prompt for, receive, and set the values for the `secondService` object by placing double slashes at the start of their lines, as shown below. By commenting out these lines, you change the program so that the user does not enter values for the `secondService` object. Instead, the values assigned by the constructor are the final values for the object.

```
//      keyboard.nextLine();  
//      System.out.print("Enter service >> ");  
//      service = keyboard.nextLine();  
//      System.out.print("Enter price >> ");  
//      price = keyboard.nextDouble();  
//      secondService.setServiceDescription(service);  
//      secondService.setPrice(price);
```

5. Save the file, and then compile and execute it. Figure 3-33 shows a typical execution. The `firstService` object contains values supplied by the user, but the `secondService` object shows the values assigned during the object's construction.



```
C:\Java>java CreateSpaServices  
Enter service >> aromatherapy  
Enter price >> 19.99  
First service details:  
aromatherapy $19.99  
Second service details:  
XXX $0.0  
C:\Java>
```

Figure 3-33 Typical execution of `CreateSpaServices` program that uses constructor values for the second object

Understanding That Classes Are Data Types

The classes that you create become data types. Programmers sometimes refer to classes as **abstract data types**, or **ADTs**. An abstract data type is a type whose implementation is hidden and accessed through its public methods. A class can also be called a **programmer-defined data type**; in other words, it is a type that is not built into the language. A class is a composite type—that is, a class is composed from smaller parts.

Java's primitive types are not composite. Java has eight built-in primitive data types such as `int` and `double`. Primitive types can also be called *scalar* types. You do not have to define these simple types; the creators of Java have already done so. For example, when the `int` type was first created, the programmers who designed it had to think of the following:

Q: What shall we call it?

A: `int`.

Q: What are its attributes?

A: An `int` is stored in four bytes; it holds whole-number values.

Q: What methods are needed by `int`?

A: A method to assign a value to a variable (for example, `num = 32;`).

Q: Any other methods?

A: Some operators to perform arithmetic with variables (for example, `num + 6`).

Q: Any other methods?

A: Of course, there are even more attributes and methods of an `int`, but these are a good start.

Your job in constructing a new data type is similar. If you need a class for employees, you should ask:

Q: What shall we call it?

A: `Employee`.

Q: What are its attributes?

A: It has an integer ID number, a `String` last name, and a `double` salary.

Q: What methods are needed by `Employee`?

A: A method to assign values to a member of this class (for example, one `Employee`'s ID number is 3232, her last name is "Walters", and her salary is 30000).

Q: Any other methods?

A: A method to display data in a member of this class (for example, display one `Employee`'s data).

Q: Any other methods?

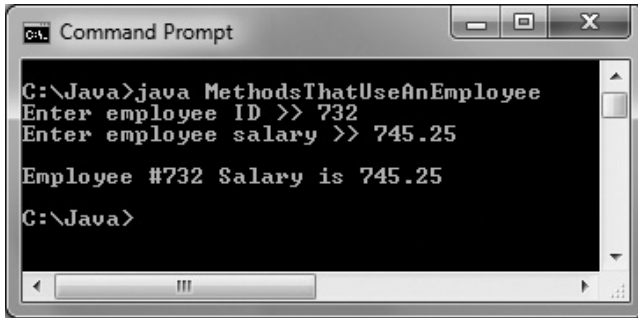
A: Probably, but this is enough to get started.

When you declare a primitive type object, you provide its type and an identifier. When you declare an object from one of your classes, you do the same. After each exists, you can use them in very similar ways. For example, suppose you declare an `int` named `myInt` and an `Employee` named `myEmployee`. Then each can be passed into a method, returned from a method, or assigned to another object of the same data type.

For example, Figure 3-34 shows a program in which the `main()` method uses two other methods. One method accepts an `Employee` as a parameter, and the other returns an `Employee`. (The `Employee` class is defined in Figure 3-28.) Figure 3-35 shows a typical execution. You can see in this sample program that an `Employee` is passed into and out of methods just like a primitive object would be. Classes are not mysterious; they are just new data types that you invent.

```
import java.util.Scanner;
class MethodsThatUseAnEmployee
{
    public static void main (String args[])
    {
        Employee myEmployee;
        myEmployee = getEmployeeData();
        displayEmployee(myEmployee);
    }
    public static Employee getEmployeeData()
    {
        Employee tempEmp = new Employee();
        int id;
        double sal;
        Scanner input = new Scanner(System.in);
        System.out.print("Enter employee ID >> ");
        id = input.nextInt();
        tempEmp.setEmpNum(id);
        System.out.print("Enter employee salary >> ");
        sal = input.nextDouble();
        tempEmp.setEmpSalary(sal);
        return tempEmp;
    }
    public static void displayEmployee(Employee anEmp)
    {
        System.out.println("\nEmployee #" + anEmp.getEmpNum() +
            " Salary is " + anEmp.getEmpSalary());
    }
}
```

Figure 3-34 The `MethodsThatUseAnEmployee` application



```
C:\Java>java MethodsThatUseAnEmployee
Enter employee ID >> 732
Enter employee salary >> 745.25

Employee #732 Salary is 745.25
C:\Java>
```

Figure 3-35 Typical execution of the `MethodsThatUseAnEmployee` application

Notice in the application in Figure 3-34 that the `Employee` declared in the `main()` method is not constructed there. An `Employee` is constructed in the `getEmployeeData()` method and passed back to the `main()` method, where it is assigned to the `myEmployee` reference. The `Employee` constructor could have been called in `main()`, but the values assigned would have been overwritten after the call to `getEmployeeData()`.

TWO TRUTHS & A LIE

Understanding That Classes Are Data Types

1. When you declare a primitive variable or instantiate an object from a class, you provide both a type and an identifier.
2. Unlike a primitive variable, an instantiated object cannot be passed into or returned from a method.
3. The address of an instantiated object can be assigned to a declared reference of the same type.

The false statement is #2. An instantiated object can be passed into or returned from a method.



You Do It

Understanding That Classes Are Data Types

In this section, you modify the `CreateSpaServices` class to include a method for data entry. This change makes the `main()` method shorter, gives you the ability to reuse code, and shows that an object of the `SpaService` class data type can be returned from a method as easily as a primitive data type.

1. Open the **CreateSpaServices.java** file if it is not still open in your text editor.
2. Delete the declarations for `service`, `price`, and `keyboard`. Declarations for these variables will now be part of the data entry method that you will create.
3. Delete the six statements that prompt the user and get values for the `firstService` objects. Also delete the seven statements that prompt the user and retrieve data for the `secondService` object. You commented out these statements in the previous “You Do It” section.
4. In place of the statements you just deleted, insert two new statements. The first sends a copy of the `firstService` object to a method named `getData()`. The method returns a `SpaService` object that will be filled with appropriate data, and this object is assigned to `firstService`. The second statement does the same thing for `secondService`.

```
firstService = getData(firstService);
secondService = getData(secondService);
```

5. After the closing curly brace for the `main()` method, but before the closing curly brace for the class, start the following `public static getData()` method. The header indicates that the method both accepts and returns a `SpaService` object. Include the opening curly brace for the method, and make declarations for `service`, `price`, and `keyboard`.

```
public static SpaService getData(SpaService s)
{
    String service;
    double price;
    Scanner keyboard = new Scanner(System.in);
```

(continues)

(continued)

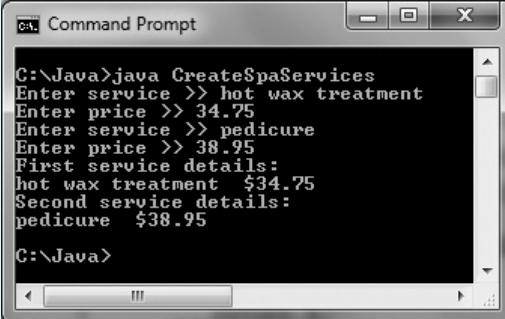
6. Continue the method by prompting the user for and accepting a service and its price. Include a final call to `nextLine()` so that the input buffer is cleared after the last numeric entry.

```
System.out.print("Enter service >> ");
service = keyboard.nextLine();
System.out.print("Enter price >> ");
price = keyboard.nextDouble();
keyboard.nextLine();
```

7. Finish the method by assigning the entered service and price to the `SpaService` object parameter using the `SpaService` class's `setServiceDescription()` and `setPrice()` methods. Then return the full object to the `main()` method, where it is assigned to the object used in the method call. Add a closing curly brace for the method.

```
s.setServiceDescription(service);
s.setPrice(price);
return s;
}
```

8. Save the file, compile it, and execute it. Figure 3-36 shows a typical execution. The execution is no different from the original version of the program, but by creating a method that accepts an unfilled `SpaService` object and returns one filled with data, you have made the `main()` method shorter and reused the data entry code.



```
C:\Java>java CreateSpaServices
Enter service >> hot wax treatment
Enter price >> 34.75
Enter service >> pedicure
Enter price >> 38.95
First service details:
hot wax treatment $34.75
Second service details:
pedicure $38.95

C:\Java>
```

Figure 3-36 Typical execution of the `CreateSpaServices` program that uses a data entry method

Don't Do It

- Don't place a semicolon at the end of a method header. After you get used to putting semicolons at the end of every statement, it's easy to start putting them in too many places. Method headers never end in a semicolon.
- Don't think “default constructor” means only the automatically supplied constructor. Although a class's automatically supplied constructor is a default constructor, so is any constructor you create that accepts no parameters.
- Don't think that a class's methods must accept its own fields' values as parameters or return values to its own fields. When a class contains both fields and methods, each method has direct access to every field within the class.
- Don't create a class method that has a parameter with the same identifier as a class field—yet. If you do, you will only be allowed to access the local variable within the method, and you will not be able to access the field. You will be able to use the same identifier and still access both values after you read the next chapter. For now, make sure that the parameter in any class method has a different identifier from any class field.

Key Terms

A **method** is a program module that contains a series of statements that carry out a task.

When you **invoke** or **call** a method, you execute it.

The **calling method** makes a **method call** that invokes the **called method**.

A **client method** is a method that calls another.

Abstraction is the programming feature that allows you to use a method name to encapsulate a series of statements.

The **method header** is the first line of the method and contains information about how other methods can interact with it.

A **declaration** is another name for a method header.

A **method body** is the set of statements between curly braces that follow the header and that carry out the method's actions.

Implementation describes the actions that execute within a method—the method body.

A **stub** is a method that contains no statements; programmers create stubs as temporary placeholders during the program development process.

Access modifier is sometimes used as another term for *access specifier*.

A **return type** indicates the type of data that, upon completion of the method, is sent back to its calling method.

To **return a value** is to send the value from a called method back to the calling method.

A **fully qualified identifier** includes a class name and a dot before the identifier.

Arguments are data items sent to methods in a method call.

Parameters are the data items received by a method.

Implementation hiding is a principle of object-oriented programming that describes the encapsulation of method details within a class.

The **interface** to a method includes the method's return type, name, and arguments. It is the part that a client sees and uses.

A **black box** is a device you can use without understanding how it works.

A **local variable** is known only within the boundaries of a method.

A method's **signature** is the combination of the method name and the number, types, and order of arguments.

Actual parameters are the arguments in a method call.

Formal parameters are the variables in a method declaration that accept the values from actual parameters.

A **return statement** ends a method and frequently sends a value from a called method back to the calling method.

A **method's type** is its return type.

Unreachable statements are those that cannot be executed because the logical path can never encounter them; an unreachable statement causes a compiler error.

Dead code is a set of statements that are logically unreachable.

An **is-a relationship** is the relationship between an object and the class of which it is a member.

An **instantiation** of a class is an object; in other words, it is one tangible example of a class.

A **class client** or **class user** is an application or class that instantiates objects of another prewritten class.

Classes can be **extended**, or used as a basis for any other class.

Data fields are data variables declared in a class outside of any method.

The **instance variables** of a class are its data components.

Assigning **private access** to a field means that no other classes can access the field's values, and only methods of the same class are allowed to set, get, or otherwise use private variables.

Information hiding is the object-oriented programming principle used when creating private access for data fields; a class's private data can be changed or manipulated only by a class's own methods and not by methods that belong to other classes.

Mutator methods set values.

Accessor methods retrieve values.

Nonstatic methods, those methods used with object instantiations, are called **instance methods**.

A **primary key** is a unique identifier for data within a database.

The **new operator** allocates the memory needed to hold an object.

A **reference to an object** is the name for a memory address where the object is held.

A **constructor** is a method that establishes an object.

A **default constructor** is one that requires no parameters; if you do not write one, a default constructor is created for a class automatically by the Java compiler.

An **abstract data type** is a type whose implementation is hidden and accessed through its public methods.

A **programmer-defined data type** is one that is created by a programmer and not built into the language.

Chapter Summary

- A method is a series of statements that carry out a task. Any method can call, or invoke, another. You place a method within a class outside of any other methods.
- Methods must include a declaration (or header or definition), an opening curly brace, a body, and a closing curly brace. A method declaration contains optional access specifiers, the return type for the method, the method name, an opening parenthesis, an optional list of parameters, and a closing parenthesis.
- When you write the method declaration for a method that can receive a parameter, you need to include the parameter type and a local name for the parameter within the method declaration parentheses. You can pass multiple arguments to methods by listing the arguments separated by commas within the call to the method. The arguments you send to the method must match (both in number and in type) the parameters listed in the method declaration.
- The return type for a method (the method's type) can be any Java type, including `void`. You use a `return` statement to send a value back to a calling method.
- Objects are concrete instances of classes. Objects gain their attributes from their classes, and all objects have predictable attributes because they are members of certain classes. In addition to their attributes, objects have methods associated with them, and every object that is an instance of a class is assumed to possess the same methods.
- A class header contains an optional access specifier, the keyword `class`, and any legal identifier you choose for the name of your class. A class contains fields, which are frequently private, and methods, which are frequently public.
- Nonstatic instance methods operate uniquely for every object. Programmers can organize their classes in many ways. Fields can be placed before or after methods, and methods can be placed in any logical order.

- To create an object that is an instance of a class, you supply a type and an identifier, and then you allocate computer memory for that object using the new operator and the class constructor. With well-written object-oriented programming methods, using implementation hiding—or the encapsulation of method details within a class—means that the calling method needs to understand only the interface to the called method.
- A constructor establishes an object and provides specific initial values for the object's data fields. A constructor always has the same name as the class of which it is a member. By default, numeric fields are set to 0 (zero), character fields are set to Unicode '\u0000', Boolean fields are set to false, and object type fields are set to null.
- A class is an abstract, programmer-defined data type, similar to Java's built-in, primitive data types.

Review Questions

1. In Java, methods must include all of the following except _____.
 - a. a declaration
 - b. a call to another method
 - c. curly braces
 - d. a body
2. All method declarations contain _____.
 - a. the keyword static
 - b. one or more explicitly named access specifiers
 - c. arguments
 - d. parentheses
3. A public static method named computeSum() is located in classA. To call the method from within classB, use the statement _____.
 - a. computeSum(classB);
 - b. classB(computeSum());
 - c. classA.computeSum();
 - d. You cannot call computeSum() from within classB.
4. Which of the following method declarations is correct for a static method named displayFacts() if the method receives an int argument?
 - a. public static int displayFacts()
 - b. public void displayFacts(int data)
 - c. public static void displayFacts(int data)
 - d. Two of these are correct.

5. The method with the declaration `public static int aMethod(double d)` has a method type of _____.
 - a. `static`
 - b. `int`
 - c. `double`
 - d. You cannot determine the method type.
6. Which of the following is a correct call to a method declared as `public static void aMethod(char code)`?
 - a. `void aMethod();`
 - b. `void aMethod('V');`
 - c. `aMethod(char 'M');`
 - d. `aMethod('Q');`
7. A method is declared as `public static void showResults(double d, int i)`. Which of the following is a correct method call?
 - a. `showResults(double d, int i);`
 - b. `showResults(12.2, 67);`
 - c. `showResults(4, 99.7);`
 - d. Two of these are correct.
8. The method with the declaration `public static char procedure(double d)` has a method type of _____.
 - a. `public`
 - b. `static`
 - c. `char`
 - d. `double`
9. The method `public static boolean testValue(int response)` returns _____.
 - a. a `boolean` value
 - b. an `int` value
 - c. no value
 - d. You cannot determine what is returned.
10. Which of the following could be the last legally coded line of a method declared as `public static int getVal(double sum)`?
 - a. `return;`
 - b. `return 77;`
 - c. `return 2.3;`
 - d. Any of these could be the last coded line of the method.

11. The nonstatic data components of a class often are referred to as the _____ of that class.
 - a. access types
 - b. instance variables
 - c. methods
 - d. objects

12. Objects contain methods and data items, which are also known as _____.
 - a. fields
 - b. functions
 - c. themes
 - d. instances

13. You send messages or information to an object through its _____.
 - a. fields
 - b. methods
 - c. classes
 - d. type

14. A program or class that instantiates objects of another prewritten class is a(n) _____.
 - a. class client
 - b. superclass
 - c. object
 - d. patron

15. The body of a class is always written _____.
 - a. in a single line, as the first statement in a class
 - b. within parentheses
 - c. between curly braces
 - d. as a method call

16. Most class data fields are _____.
 - a. `private`
 - b. `public`
 - c. `static`
 - d. `final`

17. The concept of allowing a class's private data to be changed only by a class's own methods is known as _____.
- structured logic
 - object orientation
 - information hiding
 - data masking
18. Suppose you declare an object as `Book thisBook;`. Before you store data in `thisBook`, you _____.
- also must explicitly allocate memory for it
 - need not explicitly allocate memory for it
 - must explicitly allocate memory for it only if it has a constructor
 - can declare it to use no memory
19. If a class is named `Student`, the class constructor name is _____.
- any legal Java identifier
 - any legal Java identifier that begins with *S*
 - `StudentConstructor`
 - `Student`
20. If you use the automatically supplied default constructor when you create an object, _____.
- numeric fields are set to 0 (zero)
 - character fields are set to blank
 - Boolean fields are set to `true`
 - All of these are true.

Exercises



Programming Exercises

1. Suppose that you have created a program with only the following variables:

```
int v = 4;  
int w = 6;  
double x = 2.2;
```

Suppose that you also have a method with the following header:

```
public static void calculate(int x, double y)
```

Which of the following method calls are legal?

- a. `calculate(v, w);`
 - b. `calculate(v, x);`
 - c. `calculate(x, y);`
 - d. `calculate(18, x);`
 - e. `calculate(1.1, 2.2);`
 - f. `calculate(5, 7);`
2. Suppose that a class named `ClassA` contains a private nonstatic integer named `b`, a public nonstatic integer named `c`, and a public static integer named `d`. Which of the following are legal statements in a class named `ClassB` that has instantiated an object as `ClassA obA = new ClassA();`?
- a. `obA.b = 12;`
 - b. `obA.c = 5;`
 - c. `obA.d = 23;`
 - d. `ClassA.b = 4;`
 - e. `ClassA.c = 33;`
 - f. `ClassA.d = 99;`
3. a. Create an application named `ArithmeticMethods` whose `main()` method holds two integer variables. Assign values to the variables. In turn, pass each value to methods named `displayNumberPlus10()`, `displayNumberPlus100()`, and `displayNumberPlus1000()`. Create each method to perform the task its name implies. Save the application as **ArithmeticMethods.java**.
- b. Modify the `ArithmeticMethods` class to accept the values of the two integers from a user at the keyboard. Save the file as **ArithmeticMethods2.java**.
4. a. Create an application named `Percentages` whose `main()` method holds two `double` variables. Assign values to the variables. Pass both variables to a method named `computePercent()` that displays the two values and the value of the first number as a percentage of the second one. For example, if the numbers are 2.0 and 5.0, the method should display a statement similar to “2.0 is 40% of 5.0.” Then call the method a second time, passing the values in reverse order. Save the application as **Percentages.java**.
- b. Modify the `Percentages` class to accept the values of the two `doubles` from a user at the keyboard. Save the file as **Percentages2.java**.
5. When gasoline is \$100 per barrel, then the consumer’s price at the pump is between \$3.50 and \$4.00 per gallon. Create a class named `GasPrices`. Its `main()` method holds an integer variable named `pricePerBarrel` to which you will assign a value entered by a user at the keyboard. Create a method to which you pass `pricePerBarrel`. The method displays the range of possible prices per gallon. For example, if gas is \$120 per barrel, then the price at the pump should be between \$4.20 and \$4.80. Save the application as **GasPrices.java**.
6. There are 2.54 centimeters in an inch, and there are 3.7854 liters in a U.S. gallon. Create a class named `MetricConversion`. Its `main()` method accepts an integer value from a user at the keyboard, and in turn passes the entered value to two methods. One converts the value from inches to centimeters and the other converts the same value from gallons to liters. Each method displays the results with appropriate explanation. Save the application as **MetricConversion.java**.

7. Assume that a gallon of paint covers about 350 square feet of wall space. Create an application with a `main()` method that prompts the user for the length, width, and height of a rectangular room. Pass these three values to a method that does the following:
- Calculates the wall area for a room
 - Passes the calculated wall area to another method that calculates and returns the number of gallons of paint needed
 - Displays the number of gallons needed
 - Computes the price based on a paint price of \$32 per gallon, assuming that the painter can buy any fraction of a gallon of paint at the same price as a whole gallon
 - Returns the price to the `main()` method

The `main()` method displays the final price. For example, the cost to paint a 15- by-20-foot room with 10-foot ceilings is \$64. Save the application as **PaintCalculator.java**.

8. The Harrison Group Life Insurance company computes annual policy premiums based on the age the customer turns in the current calendar year. The premium is computed by taking the decade of the customer's age, adding 15 to it, and multiplying by 20. For example, a 34-year-old would pay \$360, which is calculated by adding the decades (3) to 15, and then multiplying by 20. Write an application that prompts a user for the current year and a birth year. Pass both to a method that calculates and returns the premium amount, and then display the returned amount. Save the application as **Insurance.java**.
9. Write an application that calculates and displays the weekly salary for an employee. The `main()` method prompts the user for an hourly pay rate, regular hours, and overtime hours. Create a separate method to calculate overtime pay, which is regular hours times the pay rate plus overtime hours times 1.5 times the pay rate; return the result to the `main()` method to be displayed. Save the program as **Salary.java**.
10. Write an application that calculates and displays the amount of money a user would have if his or her money could be invested at 5 percent interest for one year. Create a method that prompts the user for the starting value of the investment and returns it to the calling program. Call a separate method to do the calculation, and return the result to be displayed. Save the program as **Interest.java**.
11. a. Create a class named **Sandwich**. Data fields include a `String` for the main ingredient (such as "tuna"), a `String` for bread type (such as "wheat"), and a `double` for price (such as 4.99). Include methods to get and set values for each of these fields. Save the class as **Sandwich.java**.
- b. Create an application named **TestSandwich** that instantiates one **Sandwich** object and demonstrates the use of the set and get methods. Save this application as **TestSandwich.java**.

12.
 - a. Create a class named `Student`. A `Student` has fields for an ID number, number of credit hours earned, and number of points earned. (For example, many schools compute grade point averages based on a scale of 4, so a three-credit-hour class in which a student earns an A is worth 12 points.) Include methods to assign values to all fields. A `Student` also has a field for grade point average. Include a method to compute the grade point average field by dividing points by credit hours earned. Write methods to display the values in each `Student` field. Save this class as **`Student.java`**.
 - b. Write a class named `ShowStudent` that instantiates a `Student` object from the class you created and assign values to its fields. Compute the `Student` grade point average, and then display all the values associated with the `Student`. Save the application as **`ShowStudent.java`**.
 - c. Create a constructor for the `Student` class you created. The constructor should initialize each `Student`'s ID number to 9999, his or her points earned to 12, and credit hours to 3 (resulting in a grade point average of 4.0). Write a program that demonstrates that the constructor works by instantiating an object and displaying the initial values. Save the application as **`ShowStudent2.java`**.
13.
 - a. Create a class named `BankAccount` with fields that hold an account number, the owner's name, and the account balance. Include a constructor that initializes each field to appropriate default values. Also include methods to get and set each of the fields. Include a method named `deductMonthlyFee()` that reduces the balance by \$4.00. Include a static method named `explainAccountPolicy()` that explains that the \$4 service fee will be deducted each month. Save the class as **`BankAccount.java`**.
 - b. Create a class named `TestBankAccount` whose `main()` method declares four `BankAccount` objects. Call a `getData()` method three times. Within the method, prompt a user for values for each field for a `BankAccount`, and return a `BankAccount` object to the `main()` method where it is assigned to one of `main()`'s `BankAccount` objects. Do not prompt the user for values for the fourth `BankAccount` object, but let it continue to hold the default values. Then, in `main()`, pass each `BankAccount` object in turn to a `showValues()` method that displays the data, calls the method that deducts the monthly fee, and displays the balance again. The `showValues()` method also calls the method that explains the deduction policy. Save the application as **`TestBankAccount.java`**.
14.
 - a. Create a class named `Painting` that contains fields for a painting's title, artist, medium (such as water color), price, and gallery commission. Create a constructor that initializes each field to an appropriate default value, and create instance methods that get and set the fields for title, artist, medium, and price. The gallery commission field cannot be set from outside the class; it is computed as 20 percent of the price each time the price is set. Save the class as **`Painting.java`**.

- b. Create a class named `TestPainting` whose `main()` method declares three `Painting` items. Create a method that prompts the user for and accepts values for two of the `Painting` objects, and leave the third with the default values supplied by the constructor. Then display each completed object. Finally, display a message that explains the gallery commission rate. Save the application as **TestPainting.java**.



Debugging Exercises

1. Each of the following files saved in the `Chapter03` folder in your downloadable student files has syntax and/or logic errors. In each case, determine and fix the problem. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, `DebugThree1.java` will become `FixDebugThree1.java`.
 - a. `DebugThree1.java`
 - b. `DebugThree2.java`
 - c. `DebugThree3.java`
 - d. `DebugThree4.java`



When you change a filename, remember to change every instance of the class name within the file so that it matches the new filename. In Java, the filename and class name must always match.



Game Zone

1. Playing cards are used in many computer games, including versions of such classics as solitaire, hearts, and poker. Design a `Card` class that contains a character data field to hold a suit (*s* for spades, *h* for hearts, *d* for diamonds, or *c* for clubs) and an integer data field for a value from 1 to 13. (When you learn more about string handling in the chapter *Characters, Strings, and the StringBuilder*, you can modify the class to hold words for the suits, such as *spades* or *hearts*, as well as words for some of the values—for example, *ace* or *king*.) Include get and set methods for each field. Save the class as **Card.java**.

Write an application that randomly selects two playing cards and displays their values. Simply assign a suit to each of the cards, but generate a random number for each card's value. Appendix D contains information on generating random numbers. To fully understand the process, you must learn more about Java classes and methods. However, for now, you can copy the following statements to generate a random number between 1 and 13 and assign it to a variable:

```
final int CARDS_IN_SUIT = 13;
myValue = ((int)(Math.random() * 100) % CARDS_IN_SUIT + 1);
```

After reading the chapter *Making Decisions*, you will be able to have the game determine the higher card. For now, just observe how the card values change as you execute the program multiple times. Save the application as **PickTwoCards.java**.



You use the `Math.random()` function to generate a random number. The function call uses only a class and method name—no object—so you know the `random()` method must be a static method.

2. Computer games often contain different characters or creatures. For example, you might design a game in which alien beings possess specific characteristics such as color, number of eyes, or number of lives. Design a character for a game, creating a class to hold at least three attributes for the character. Include methods to get and set each of the character's attributes. Save the file as **MyCharacter.java**. Then write an application in which you create at least two characters. In turn, pass each character to a display method that displays the character's attributes. Save the application as **TwoCharacters.java**.



Case Problems

1. a. Carly's Catering provides meals for parties and special events. In Chapter 2, you wrote an application that prompts the user for the number of guests attending an event, displays the company motto with a border, and then displays the price of the event and whether the event is a large one. Now modify the program so that the `main()` method contains only three executable statements that each call a method as follows:
 - The first executable statement calls a `public static int` method that prompts the user for the number of guests and returns the value to the `main()` method.
 - The second executable statement calls a `public static void` method that displays the company motto with the border.
 - The last executable statement passes the number of guests to a `public static void` method that computes the price of the event, displays the price, and displays whether the event is a large event.

Save the file as **CarlysEventPriceWithMethods.java**.

- b. Create a class to hold `Event` data for Carly's Catering. The class contains:
 - Two `public final static` fields that hold the price per guest (\$35) and the cutoff value for a large event (50 guests)
 - Three `private` fields that hold an event number, number of guests for the event, and the price. The event number is stored as a `String` because Carly plans to assign event numbers such as *M312*.

- Two `public` set methods that set the event number (`setEventNumber()`) and the number of guests (`setGuests()`). The price does not have a set method because the `setGuests()` method will calculate the price as the number of guests multiplied by the price per guest every time the number of guests is set.
- Three `public` get methods that return the values in the three nonstatic fields

Save the file as **Event.java**.

- c. Use the `CarlysEventPriceWithMethods` class you created in Step 1a as a starting point for a program that demonstrates the `Event` class you created in Step 1b, but make the following changes:
 - You already have a method that gets a number of guests from a user; now add a method that gets an event number. The `main()` method should declare an `Event` object, call the two data entry methods, and use their returned values to set the fields in the `Event` object.
 - Call the method from the `CarlysEventPriceWithMethods` class that displays the company motto with the border. The method is accessible because it is `public`, but you must fully qualify the name because it is in another class.
 - Revise the method that displays the event details so that it accepts the newly created `Event` object. The method should display the event number, and it should still display the number of guests, the price per guest, the total price, and whether the event is a large event.

Save the program as **EventDemo.java**.

2. a. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. In Chapter 2, you wrote an application that prompts the user for the number of minutes a piece of sports equipment was rented, displays the company motto with a border, and displays the price for the rental. Now modify the program so that the `main()` method contains only three executable statements that each call a method as follows:
 - The first executable statement calls a method that prompts the user for the rental time in minutes and returns the value to the `main()` method.
 - The second executable statement calls a method that displays the company motto with the border.
 - The last executable statement passes the number of minutes to a method that computes the hours, extra minutes, and price for the rental, and then displays all the details.

Save the file as **SammysRentalPriceWithMethods.java**.

- b. Create a class to hold `Rental` data for Sammy's Seashore Supplies. The class contains:
 - Two `public final static` fields that hold the number of minutes in an hour and the hourly rental rate (\$40)

- Four `private` fields that hold a contract number, number of hours for the rental, number of minutes over an hour, and the price. The contract number is stored as a `String` because Sammy plans to assign contract numbers such as *K681*.
- Two `public` set methods. One sets the contract number (`setContractNumber()`). The other is named `setHoursAndMinutes()`, and it accepts the number of minutes for the rental and then sets the hours, extra minutes over an hour, and the total price. Recall from Chapter 2 that the price is \$40 per hour plus \$1 for every extra minute.
- Four `public` get methods that return the values in the four nonstatic fields

Save the file as **Rental.java**.

- c. Use the `SammysRentalPriceWithMethods` class you created in Step 2a as a starting point for a program that demonstrates the `Rental` class you created in Step 2b, but make the following changes:
 - You already have a method that gets a number of minutes from a user; now add a method that gets a contract number. The `main()` method should declare a `Rental` object, call the two data entry methods, and use their returned values to set the fields in the `Rental` object.
 - From the `SammysRentalPriceWithMethods` class, call the `RentalDemo` method that displays the company motto with the border. The method is accessible because it is `public`, but you must fully qualify the name because it is in another class.
 - Revise the method that displays the rental details so that it accepts the newly created `Rental` object. The method should display the contract number, and it should still display the hours and minutes, the hourly rate, and the total price.

Save the program as **RentalDemo.java**.