

# USING METHODS, CLASSES, AND OBJECTS

**In this chapter, you will:**

- ◆ Create methods with no arguments, a single argument, and multiple arguments
- ◆ Create methods that return values
- ◆ Learn about class concepts
- ◆ Create a class
- ◆ Use instance methods
- ◆ Declare objects
- ◆ Organize classes
- ◆ Use constructors

**H**ow do you feel about programming so far?” asks your new mentor, Lynn Greenbrier, who is head of computer programming for Event Handlers Incorporated.

“It’s fun!” you reply. “It’s great to see something actually work, but I still don’t understand what the other programmers are talking about when they mention ‘object-oriented programming.’ I think everything is an object, and objects have methods, but I’m not really clear on this whole thing at all.”

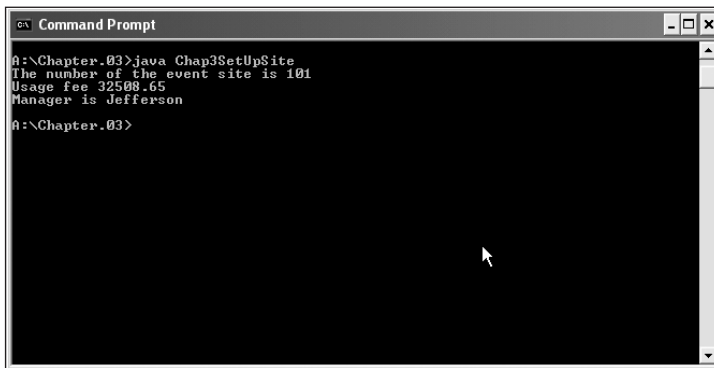
“Well then,” Lynn says, “let me explain methods, classes, and objects.”

## PREVIEWING THE SETUPSITE PROGRAM USING THE EVENTSITE CLASS

You will now preview the SetUpSite program that is saved on your Student Disk.

**To preview the SetUpSite program on your Student Disk:**

1. Start your text editor, open the **Chap3EventSite.java** file from the Chapter.03 folder on your Student Disk, and then examine the code. This file contains a class definition for a class that stores information about event sites used by Event Handlers Incorporated to host planned events.
2. Go to the command line, and then type **javac Chap3EventSite.java** to compile the Chap3EventSite.java file.
3. Use your text editor to open the **Chap3SetUpSite.java** file from the Chapter.03 folder, and then examine the code. This file contains a program that assigns values to the data regarding event sites that Event Handlers Incorporated uses to hold events. The program then displays that data on the screen. You will create a similar program in this chapter.
4. At the command line, type **javac Chap3SetUpSite.java** to compile the Chap3SetUpSite.java file.
5. Type **java Chap3SetUpSite** to execute the program. Information about an event site used by Event Handlers Incorporated will appear on the screen as shown in Figure 3-1.



```
Q:\Chapter.03>java Chap3SetUpSite
The number of the event site is 101
Usage fee 32508.65
Manager is Jefferson
Q:\Chapter.03>
```

**Figure 3-1** Output of the Chap3SetUpSite program

Although the output shown in Figure 3-1 is modest, you have just witnessed several important programming concepts in action. The Chap3EventSite file contains a class definition that represents a real-life object—a site at which Event Handlers can hold an event. The class includes methods to assign values to and get values from data fields that pertain to event sites. The Chap3SetUpSite file creates an actual site with data representing a site number, a fee, and a manager's name. You will create similar files in this chapter.

## CREATING METHODS WITH NO ARGUMENTS, A SINGLE ARGUMENT, AND MULTIPLE ARGUMENTS

A **method** is a series of statements that carry out a task. Any class can contain an unlimited number of methods. Within a class, the simplest methods you can invoke don't require any arguments or return any values. Consider the simple First Java program's First class that you saw in Chapter 1 and that appears in Figure 3-2.

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

**Figure 3-2** The First class

Suppose you want to add three additional lines of output to this program to display your company's name and address. You can simply add three new `println()` statements, but instead you might choose to create a method to display the three lines.



Although there are differences, if you have used other programming languages, you can think of methods as being similar to procedures, functions, or subroutines.

There are two major reasons to create a method to display the three lines. First, the `main()` method will remain short and easy to follow because `main()` will contain just one statement to call a 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 program 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. A method must include the following:

- A declaration (or header or definition)
- An opening curly brace
- A body
- A closing curly brace

The method declaration contains the following:

- Optional access modifiers
- The return type for the method

- The method name
- An opening parenthesis
- An optional list of method arguments (you separate the arguments with commas if there is more than one)
- A closing parenthesis

You first learned about access modifiers in Chapter 1. The access modifier for a method can be any of the following modifiers: **public**, **private**, **protected**, or **static**. Most often, methods are given **public** access. Endowing a method with **public** access means any class can use it. Additionally, like `main()`, any method that can be used from anywhere within the class (that is, any class-wide method) requires the keyword modifier **static**. Therefore, you can write the `nameAndAddress()` method shown in Figure 3-3. According to its declaration, the method is **public** and **static**. It returns nothing, so its return type is **void**. The method receives nothing, so its parentheses are empty. Its body, consisting of three `println()` statements, appears within curly braces.

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

**Figure 3-3** The `nameAndAddress()` method

You place the entire method within the program that will use it, but not within any other method. Figure 3-4 shows where you can place a method in the `First` program.

```
public class First
{
    public static void main(String[] args)
    {
        System.out.println("First java program");
    }
    |
    // You can place additional methods here,
    // outside the main() method
}
```

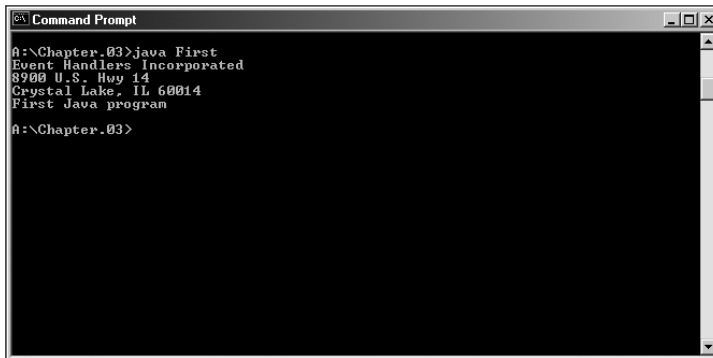
**Figure 3-4** Placement of methods

If the `main()` method calls the `nameAndAddress()` method, then you simply use the `nameAndAddress()` method's name as a statement within the body of `main()`. Figure 3-5 shows the complete program.

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

**Figure 3-5** First class calling the nameAndAddress() method

The output from the program shown in Figure 3-5 appears in Figure 3-6. Because the main() method calls the nameAndAddress() method before it prints the phrase “First Java program”, the name and address appear first in the output.



```
A:\Chapter.03>java First
Event Handlers Incorporated
8900 U.S. Hwy 14
Crystal Lake, IL 60014
First Java program
A:\Chapter.03>
```

**Figure 3-6** Output of the First program with the nameAndAddress() method

If you want to use the nameAndAddress() method in another program, one additional step is required. In the Java programming language, the new program, with its own main() method, is a different class. If you place the nameAndAddress() method within the new class, the compiler will not recognize it unless you write it as **First.nameAndAddress()**; to notify the new class that the method is located in the First class. Notice the use of the class name, followed by a dot, and then followed by the method. You have used similar syntax for the System.out.println() method.



Each of two different classes can have their own method named nameAndAddress(). Such a method in the second class would be entirely distinct from the identically named method in the first class.

Next you will create a new class named `SetUpSite`, which you will eventually use to set up one `EventSite` object. For now, the class will contain a `main()` method and a `statementOfPhilosophy()` method for Event Handlers Incorporated.

**To create the `SetUpSite` class:**

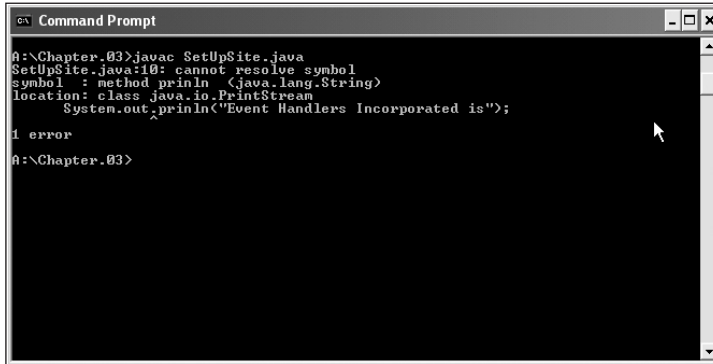
1. Open a new document in your text editor.
2. Type the following shell program to create a `SetUpSite` class and an empty `main()` method:

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

3. Place the insertion point to the right of the opening bracket in the `main()` method, press **[Enter]** to start a new line, and then type **`statementOfPhilosophy();`** between the curly braces of the `main()` method to place a call to a `statementOfPhilosophy()` method.
4. Type the following code for the `statementOfPhilosophy()` method just before the closing curly brace for the `SetUpSite` class code:

```
public static void statementOfPhilosophy()
{
    System.out.println("Event Handlers Incorporated is");
    System.out.println
        ("dedicated to making your event");
    System.out.println("a most memorable one.");
}
```

5. Save the file as **`SetUpSite.java`** in the `Chapter.03` folder on your Student Disk.
6. At the command line, compile the program by typing **`javac SetUpSite.java`**. If you receive any error messages, you must correct their cause. Figure 3-7 shows the error message received when `println()` is spelled incorrectly within the `SetUpSite.java` file. Notice the message indicates that the file is `SetUpSite.java`, the line on which the error occurs is line 10, and the error is “method `println`...not found...” To help you, Java displays the offending line, and a caret appears just below and after the word that the compiler doesn’t understand. To correct the spelling error, you return to the `SetUpSite.java` file, fix the mistake, save the file, and then compile it again.

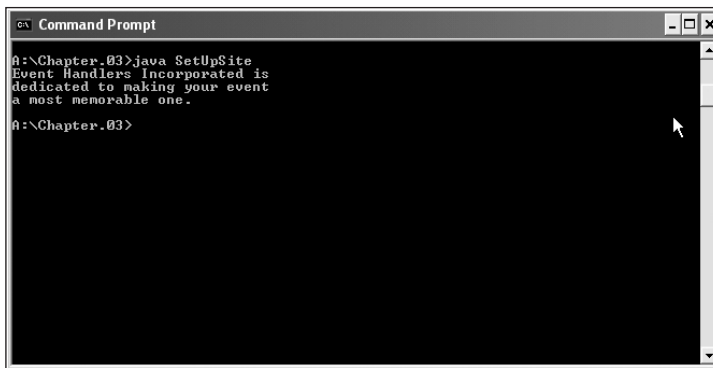


```
A:\Chapter.03>javac SetUpSite.java
SetUpSite.java:10: cannot resolve symbol
symbol : method println (java.lang.String)
location: class java.io.PrintStream
System.out.println("Event Handlers Incorporated is");

1 error
A:\Chapter.03>
```

Figure 3-7 SetUpSite program with a syntax error

7. Execute the program using the command **java SetUpSite**. Your output should look like Figure 3-8.



```
A:\Chapter.03>java SetUpSite
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
A:\Chapter.03>
```

Figure 3-8 Output of the SetUpSite program



If you want one class to call the method of another class, both classes should reside in the same folder. If they are not saved in the same place, your compiler will issue the error message, "undefined variable or class name."

Next you will see how to call the `statementOfPhilosophy()` method from another class.

### To call a method from another class:

1. First, open a new document in your text editor, and then enter the program that appears in Figure 3-9.

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

**Figure 3-9** TestStatement program

2. Save the file as **TestStatement.java** in the Chapter.03 folder on your Student Disk.
3. Compile the program with the command **javac TestStatement.java**.



If necessary, correct any errors, save the file, and then repeat Step 3 to compile the file again.

4. Execute the program with the command **java TestStatement**. Your output should look like Figure 3-10.

```
Command Prompt
A:\Chapter.03>java TestStatement
Calling method from another class:
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
A:\Chapter.03>
```

**Figure 3-10** Output of the TestStatement program

It is also possible to revise `SetUpSite.java` by removing the `main()` method and the `statementOfPhilosophy()` call from the `SetUpSite.java` text document. The remaining text would appear as shown in Figure 3-11, with only the `statementOfPhilosophy()` method remaining.



**To revise the SetUpSite program:**

1. Remove the `main()` method and the `statementOfPhilosophy()` call from the `SetUpSite.java` text document.
2. Save the revised program as **SetUpSite2.java**.
3. Type **`javac SetUpSite2.java`** and compile the program as the `SetUpSite2.class`. Because there is no longer a `main()` method, you can't run the revised program by typing **`java SetUpSite2`**.

```
public class SetUpSite2
{
    public static void statementOfPhilosophy()
    {
        System.out.println("Event Handlers Incorporated is");
        System.out.println("dedicated to making your event");
        System.out.println("a most memorable one.");
    }
}
```

**Figure 3-11** SetUpSite2 program

You can also include the new `SetUpSite2` program as a class within the `TestStatement` program.

**To add the SetUpSite2 program to TestStatement.java:**

1. Open the **TestStatement.java** text document in your text editor, and then change the class name to **TestStatement2**.
2. Open the **SetUpSite2** text document containing the `StatementOfPhilosophy()` method in a separate text window.
3. Click **Edit** in the **SetUpSite2** text window menu, and then click **Select All** to select the **SetUpSite2** class text.
4. Click **Edit** and then click **Copy** to copy the **SetUpSite2** class text.
5. Switch to the open **TestStatement2.java** text document, place the insertion point in front of `public classTestStatement2` and then press **[Enter]** to create a blank line above `public classTestStatement2`.
6. Place the insertion point at the beginning of the blank line, click **Edit**, and then click **Paste** to paste the **SetUpSite2** class text.
7. Position the insertion point on the keyword **public** and delete it.
8. Save the program as **TestStatement2.java**.
9. Type the command **`javac TestStatement2.java`**.

10. Type the command **java TestStatement2**. Your revised TestStatement2 program should look like Figure 3-12. The output for this program is shown in Figure 3-13.

```
class SetUpSite2
{
    public static void statementOfPhilosophy()
    {
        System.out.println("Event Handlers Incorporated is");
        System.out.println("dedicated to making your event");
        System.out.println("a most memorable one.");
    }
}
public class TestStatement2
{
    public static void main(String[] args)
    {
        System.out.println
        ("Calling method from another class:");
        SetUpSite.statementOfPhilosophy();
    }
}
```

**Figure 3-12** Revised TestStatement2 program



Because you must name a file from a class exactly the same as the public class name, you must remove the public class modifier from the SetUpSite2 class. If you don't, you will receive an error message. The compiler will not know which name to assign to the class file from two class programs if each program has a public class modifier.

```

C:\> java TestStatement2
Calling method from another class:
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
C:\>
```

**Figure 3-13** Output of the TestStatement2 program

## Creating Methods that Require a Single Argument

Some methods require additional information. If a method could not receive your communications, called **arguments**, then you would have to write an infinite number of methods to cover every possible situation. For 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 method, which is then carried out in the same manner, no matter what date and time are involved. 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, a `square2()` method, and so on.

An important principle of object-oriented programming is the notion of implementation hiding. When you make a request to a method, you don't know the details of how the method is executed. For example, when you make a reservation, you do not need to know how the reservation is actually made 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. With well-written object-oriented programming methods, **implementation hiding** allows that the invoking program must know the name of the method and what type of information to send it (and what type of return to expect), but the program does not need to know how the method works. Additionally, you can substitute a new, improved method and, as long as the interface to the method does not change, you won't need to make any changes in programs that invoke the method.



At any call, the `println()` method can receive any one of an infinite number of arguments. No matter what message is sent to `println()`, the message displays correctly.



Hidden implementation methods are often referred to as existing in a black box.

When you write the method declaration for a method that can receive an argument, you must include the following items within the method declaration parentheses:

- The type of the argument
- A local name for the argument

For example, the declaration for a public method named `predictRaise()` that displays a person's salary plus a 10 percent raise could have the declaration `public void predictRaise(double moneyAmount)`. You can think of the parentheses in a

method declaration as a funnel into the method—data arguments listed there are “dropped in” to the method.

The argument `double moneyAmount` within the parentheses indicates that the `predictRaise()` method will receive a figure of type `double`. Within the method, the figure (or salary amount) will be known as `moneyAmount`. Figure 3-14 shows a complete method.

```
public void predictRaise(double moneyAmount)
{
    double newAmount;
    newAmount = moneyAmount * 1.10;
    System.out.println
        ("With raise salary is " + newAmount);
}
```

**Figure 3-14** The `predictRaise()` method

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

Within a program, 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` value and assigned an appropriate value. 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 `moneyAmount` within the method. The identifier `moneyAmount` holds any `double` value passed into the method. It’s interesting to note that if the value in the method call is a variable, it might possess the same identifier as `moneyAmount`, or a different one, such as `mySalary`. The identifier `moneyAmount` is simply a placeholder while it is being used within the method, no matter what name it “goes by” in the calling program.



The variable `moneyAmount` is a local variable to the `predictRaise()` method.

If a programmer changes the way in which the 10 percent raise is calculated—for example, by coding `newAmount = moneyAmount + (moneyAmount * .10);`—no program that uses the `predictRaise()` method will ever know the difference. The program will pass a value into `predictRaise()` and then a calculated result will appear on the screen.

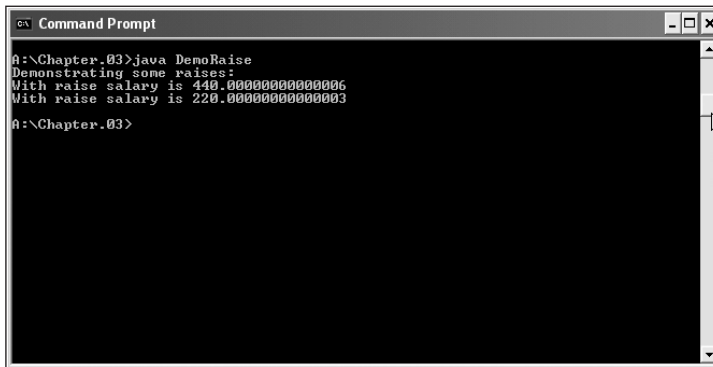
Figure 3-15 shows a complete program that uses the `predictRaise()` method twice. The program’s output appears in Figure 3-16.

```

public class DemoRaise
{
    public static void main(String[] args)
    {
        double mySalary = 200.00;
        System.out.println("Demonstrating some raises:");
        predictRaise(400.00);
        predictRaise(mySalary);
    }
    public static void predictRaise(double moneyAmount)
    {
        double newAmount;
        newAmount = moneyAmount * 1.10;
        System.out.println("With raise salary is " +
            newAmount);
    }
}

```

**Figure 3-15** Complete program using the predictRaise() method twice



```

A:\Chapter.03>java DemoRaise
Demonstrating some raises:
With raise salary is 440.00000000000006
With raise salary is 220.00000000000003
A:\Chapter.03>

```

**Figure 3-16** Output of the DemoRaise program



Notice the output in Figure 3-16. Floating-point arithmetic is always somewhat imprecise.

## Creating Methods that Require Multiple Arguments

A method can require more than one argument. 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 creating 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 3-17 shows a method that uses two such arguments.

```
public void predictRaiseGivenIncrease
(double moneyAmount, double percentRate)
{
    double newAmount;
    newAmount = moneyAmount * (1 + percentRate);
    System.out.println("With raise salary is " +
        newAmount);
}
```

**Figure 3-17** The `predictRaiseGivenIncrease()` method



Note that a declaration for a method that receives two or more arguments must list the type for each argument separately, even if the arguments have the same type.

In the header of the `predictRaiseGivenIncrease()` method, the arguments in parentheses are shown on separate lines to fit in this book's margin space. You could place the parentheses and arguments on the same line as the function header.

In Figure 3-17, two arguments (`double moneyAmount` and `double percentRate`) appear within the parentheses in the method header. The arguments are separated by a comma, and each argument requires its own named type (in this case, both are `double`) as well as an identifier. When values are passed to the method in a statement such as `predictRaiseGivenIncrease(mySalary,promisedRate);`, the first value passed will be referenced as `moneyAmount` within the method, and the second value passed will be referenced as `percentRate`. Therefore, it is very important that arguments passed to the method be passed in the correct order. The call `predictRaiseGivenIncrease(200.00,.10);` results in output representing a 10 percent raise based on a \$200 salary amount (or \$220), but `predictRaiseGivenIncrease(.10,200.00);` results in output representing a 200 percent raise based on a salary of 10 cents (or \$20.10).



If two method arguments are the same type—for example, two doubles—passing them to a method in the wrong order results in a logical error. If a method expects arguments of diverse types, then passing arguments in reverse order constitutes a syntax error.

You can write a method so that it takes any number of arguments in any order. However, when you call a method, the arguments you send to a method must match—both in number and in type—the arguments listed in the method declaration. Thus, a method to compute an automobile salesperson's commission amount might require arguments such as an integer value of a car sold, a double percentage commission rate, and a character code for the vehicle type. The correct method will execute only when three arguments of the correct types are sent in the correct order.



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 the formal parameters.

## CREATING METHODS THAT RETURN VALUES

The return type for a method can be any type used in the Java programming language, which includes the primitive (or scalar) 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 is written `public static void nameAndAddress()`. This method is public and it returns no value, so it is type `void`. A method that returns `true` or `false`, depending on whether or not an employee worked overtime hours might be `public Boolean overtime()`. This method is public and it returns a Boolean value, so it is type `Boolean`.



In addition to returning the primitive types, a method can return a class type. If a class named `BankLoan` exists, a method might return an instance of a `BankLoan`, as in `public BankLoan approvalProcess()`. In other words, a method can return anything from a simple `int` to a complicated `BankLoan` with 20 data fields.

The header for a method that displays a raise amount is `public static void predictRaise(double moneyAmount)`. If you want to create a method to return the new, calculated salary value rather than display the raised salary, the header would be `public double calculateRaise(double moneyAmount)`. Figure 3-18 shows this method.

```
public void calculateRaise(double moneyAmount)
{
    double newAmount;
    newAmount = moneyAmount * 1.10;
    return newAmount;
}
```

**Figure 3-18** The `calculateRaise()` method

Notice the return type `double` in the method header. Also notice the return statement that is the last statement within the method. The **return statement** causes the value stored in `newAmount` to be sent back to any method that calls the `calculateRaise()` method.

If a method returns a value, then when you call the method, you will usually want to use the returned value, although you are not required to do so. For example, when you

invoke the `calculateRaise()` method, you might want to assign the value to a double variable named `myNewSalary`, as in `myNewSalary = calculateRaise(mySalary);`. The `calculateRaise()` method returns a double, so it is appropriate to assign the returned value to a double variable.

Alternately, you can choose to display a method's returned value directly, without storing it in any variable, as in `System.out.println("New salary is " + calculateRaise(mySalary));`. In this last statement, the call to the `calculateRaise()` method is made from within the `println()` method call. Because `calculateRaise()` returns a double, you can use the method call `calculateRaise()` in the same way that you would use any simple double value. For example, besides printing the value of `calculateRaise()`, you can perform math with it, assign it, and so on.

Next you will add a method to the `SetUpSite2` class that both receives an argument and returns a value. The purpose of the method is to take the current year and calculate how long Event Handlers Incorporated has been in business.

#### To add a method that receives an argument and returns a value:

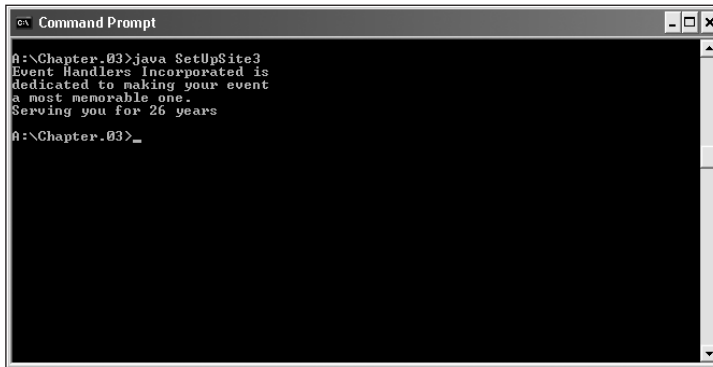
1. Open the **SetUpSite2.java** file in the text editor, and then change the class file to **SetUpSite3**.
2. Position the insertion point to the right of the opening curly brace of the `main()` method of the class, and then press **[Enter]** to start a new line.
3. Type `int currentYear = 2003;` to declare a variable to hold the current year, and then press **[Enter]**.
4. Type `int age;` to declare another variable to hold the age of Event Handlers Incorporated.
5. Position the insertion point at the end of the call to the `statementOfPhilosophy()` method in the `main()` method of the class, and then press **[Enter]** to start a new line. You will add a call to receive the current year and calculate how long Event Handlers Incorporated has been in business by subtracting the year of its inception, which is 1977.
6. Type `age = calculateAge(currentYear);` as a call to a `calculateAge()` method.
7. Press **[Enter]**, and then type `System.out.println("Serving you for " + age + " years");` to print the number of years the company has been in business. Now you will write the `calculateAge()` method.
8. Position the insertion point after the closing bracket of the `statementOfPhilosophy()` method, press **[Enter]** to start a new line before the closing bracket of the program, and then enter the method shown in Figure 3-19. The method will receive an integer value. Within the `calculateAge()` method, the value will be known as `currDate`. Note that the name `currDate` does not possess the same identifier as `currentYear`, which is the variable being passed in, although it could. Notice also that the method declaration indicates an `int` value will be returned.



```
public static int calculateAge(int currDate)
{
    int yrs;
    yrs = currDate - 1977;
    return yrs;
}
```

**Figure 3-19** The calculateAge() method

9. Save the file as **SetUpSite3.java**, compile it, and correct any errors. Execute the program and confirm that the results are correct. See Figure 3-20.



```
Q:\Chapter.03>java SetUpSite3
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
Serving you for 26 years
Q:\Chapter.03>
```

**Figure 3-20** Output of the SetUpSite3 program with the calculateAge() method

## LEARNING ABOUT CLASS CONCEPTS

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, or 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 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**, relationships that are correct only if said 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 can’t say, “My Desk *is an* oak desk with a scratch on top or 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.

The concept of a class is useful because of its reusability. Objects inherit attributes from 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, some quantity of food, and some kind of gifts. 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, what food will be served, or what gifts will be received at this particular party, but you understand that because all parties have guests and refreshments, then this one must too. Similarly, you can also apply this thinking to the stock market. Even though every stock market purchase is unique, each stock purchase must have a dollar amount and a number of shares.



The data components of a class are often referred to as the instance variables of that class. Also, class object attributes are often called fields to help distinguish them from other variables you might use.

In addition to their attributes, class 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 parties, at some point, you must set the date and time. 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 the party.

Your graduation party, then, might possess the identifier `myGraduationParty`. As a member of the Party class, `myGraduationParty`, like all parties, might have data members for the date and time methods `setDate()` and `setTime()`. When you use them, the `setDate()` and `setTime()` methods require arguments, or information passed to them. For example, `myGraduationParty.setDate("May 12")` and `myGraduationParty.setTime("6 P.M.")` invoke methods that are available for `myGraduationParty` and send it arguments. 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 `setDate()` and `setTime()` methods, the arguments you send to `yourAnniversaryParty` will be different from those you send to `myGraduationParty`. Within any object-oriented program, you are continuously making requests to objects' methods, and often including arguments as part of those requests.

Additionally, some methods used in a program 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 figure in dollars and cents, and a method within an Inventory class might return true or false, depending on the method's determination of whether or not an item is at the reorder point.

There are two parts to object-oriented programming. First, you must create the classes of objects from which objects will be instantiated, and second, you must write other classes to use the objects (and their data and their methods). The same programmer does not need to accomplish these two tasks. 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 programs. You can call a program or class that instantiates objects of another prewritten class a **class client** or **class user**.



The System class that you used in Chapter 1 is 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.

## CREATING A CLASS

When you create a class, first you must assign a name to the class, and then 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 modifier
- The keyword `class`
- Any legal identifier you choose for the name of your class

For example, a header for an `Employee` class is `public class Employee`. The keyword `public` is a class access modifier. You can use the following class access modifiers when defining a class: `public`, `final`, or `abstract`. If you do not specify an access modifier, access becomes `public`.

Public classes are accessible by all objects, which means that public classes can be **extended**, or used as a basis for any other class. The most liberal form of access is `public`. Public access means that if you develop a good `Employee` class, and some day you want to develop two more-specific classes, `SalariedEmployee` and `HourlyEmployee`, then you will not have to start from scratch. Each new class can become an extension of the original `Employee` class, inheriting its data and methods. The other access modifiers (or the omission of any access modifier) impose at least some limitations on extensibility. (You use the other access modifiers only under special circumstances.) You will use the public access modifier for most of your classes.

After writing the class header `public class Employee`, you write the body of the `Employee` class, containing its data and methods, between a set of curly braces. Figure 3-21 shows the shell for the `Employee` class.

```
public class Employee
{
    //Instance variables and methods go here
}
```

**Figure 3-21** Employee class shell

You place the instance variables, or fields, for the `Employee` class as statements within the curly braces. For example, you can declare an employee number that will be stored as an integer simply as `int empNum;`. However, programmers frequently include an access modifier for each of the class fields and declare the `empNum` as `private int empNum;`.

The allowable field modifiers are `private`, `public`, `static`, and `final`. Most class fields are `private`, which provides the highest level of security. Private access means that no other classes can access a field's values, and only methods of the same class are allowed to set, get, or otherwise use private variables. 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, most class methods are not usually `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 field modifiers are the same as the method modifiers with one addition—the `final` modifier. You will learn to use the `final` modifier in Chapter 4.

The entire class so far appears in Figure 3-22. It defines a public class named `Employee`, with one field, which is a private integer named `empNum`.

Next you will create a class to store information about event sites for Event Handlers Incorporated.

#### To create the class:

1. Open a new document in your text editor.

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

**Figure 3-22** Employee class with one data field

**3**

2. Type the following class header and the curly braces to surround the class body:

```
public class EventSite
{
}
```

3. Type **private int siteNumber;** between the curly braces to insert the private data field that will hold an integer site number for each event site used by the company.
4. Save the file as **EventSite.java** in the Chapter.03 folder on your Student Disk.
5. To ensure you have not made any typographical errors, compile the class by typing **javac EventSite.java** at the command-line prompt. If necessary, correct any errors, save your work, and then compile again. Do not execute the class.

## USING INSTANCE METHODS

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(). Figure 3-23 shows the complete getEmpNum() method.

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

**Figure 3-23** The getEmpNum() method

The getEmpNum() method contains just one statement: the statement that accesses the value of the private empNum field.

Notice that, unlike the class methods you created earlier in this chapter, the getEmpNum() method does not employ the **static** modifier. The keyword **static** is used for class-wide 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, then many of your

methods will be static so you can call them from within `main()`. However, if you are creating a class from which objects will be instantiated, most methods will probably be non-static because you will associate the methods with individual objects. Methods used with object instantiations are called **instance methods**.



You can call class methods without creating an instance of the class. Instance methods require an instantiated object.

Next you will add an instance method to the `EventSite2` class that will retrieve the value of an event site's number.

#### To add an instance method to the `EventSite2` class:

1. Open the **EventSite.java** file in your text editor and change the class name to **EventSite2**.
2. Within the `EventSite2` class's curly braces and after the declaration of the `siteNumber` field, enter the following `getSiteNumber()` method to return the site number to any calling class:

```
public int getSiteNumber()  
{  
    return siteNumber;  
}
```

3. Save the file as **EventSite2.java**.

When a class contains data fields, you want a means to assign values to the data fields. For an `Employee` class with an `empNum` field, you need a method with which to set the `empNum`. Figure 3-24 shows a method that sets the `empNum`. The method is a void method because there is no need to return any value to a calling program. The method receives an integer, locally called `emp`, to be assigned to `empNum`.

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

**Figure 3-24** The `setEmpNum()` method

Next you will add a `setSiteNumber()` method to the `EventSite2` class. This method takes an integer argument and assigns it to the `siteNumber` of an `EventSite` object.

#### To add a method to the `EventSite3` class:

1. Open the **EventSite2.java** program, change the class name to **EventSite3**, then add the following method to the **EventSite3.java** file after the final curly brace for the `getSiteNumber()` method, but prior to the closing curly brace for the `EventSite3` class:

```
public void setSiteNumber(int n)
{
    siteNumber = n;
}
```

The argument *n* represents any number sent to this method.

2. Save the file as **EventSite3.java**, compile it, and then correct any syntax errors. (You cannot run this file as a program.)

## DECLARING OBJECTS

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 which 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 define an integer as `int someValue;` and you might define an Employee as `Employee someEmployee;`, where `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 only for primitive type variables. To allocate the needed memory, you must use the **new operator**. After you define `someEmployee` with the `Employee someEmployee;` statement, the statement that actually sets aside enough memory to hold a `someEmployee = new Employee();`.

You can also define and reserve memory for `someEmployee` in one statement, as in `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. The equals sign is the assignment operator, so a value is being assigned to `someEmployee`. 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 will locate it at the appropriate address for you—but `someEmployee` does need to know its own address.



Every object name is also a reference—that is, a computer memory location.

The last portion of the statement, `Employee()`, with its parentheses, looks suspiciously like a method name. In fact, it is the name of a method that constructs an `Employee` object. `Employee()` is a **constructor method**. You will write your own constructor methods later in this section, but when you don't write a constructor method for a class object, Java writes one for you, and the name of the constructor method is always the same as the name of the class whose objects it constructs.

Next you will instantiate an `EventSite3` object.

**To instantiate an object:**

1. Open the **SetUpSite3.java** file from the Chapter.03 folder in your text editor. Change the class name to **SetUpSite4**.
2. Place the insertion point at the end of `int age` within the `main()` method, press **[Enter]** to start a new line, and then type **`EventSite3 oneSite = new EventSite3();`** to allocate memory for a new `EventSite4` object named `oneSite`.
3. Save the file as **SetUpSite4.java** and then compile it. If necessary, correct any errors, and save and compile again.

After an object has been instantiated, its methods can be accessed using the object's identifier, a dot, and a method call. For example, if an `Employee` class method to change a salary is written using the code in Figure 3-25, and an `Employee` was declared with `Employee clerk = new Employee();`, then the clerk's salary can be changed to 350.00 with the call `clerk.changeSalary(350.00);`. The method `changeSalary()` is applied to the object `clerk`, and the argument 350.00 (a double type value) is passed to the method.

```
public void changeSalary(double newAmount)
{
    salary = newAmount;
}
```

**Figure 3-25** The `changeSalary()` method

Within the same program, the statements `Employee secretary = new Employee();` and `secretary.changeSalary(420.00);` would apply the same `changeSalary()` method, but using a different argument value, to different objects that belong to the same class.

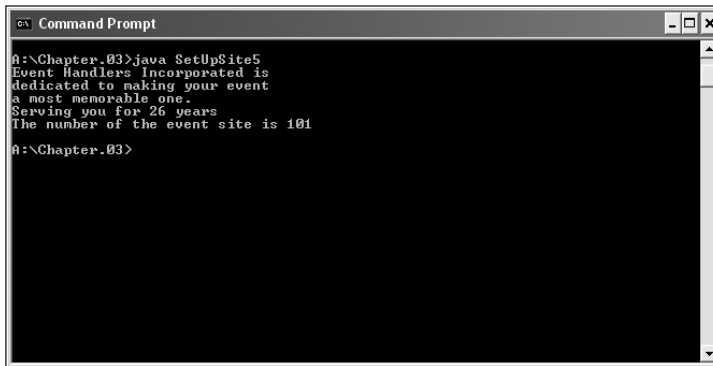
Next you will add calls to the `getSiteNumber()` and `setSiteNumber()` methods for the `oneSite` object member of the `EventSite3` class.

To add the calls to the methods for the `oneSite` object member:

1. Open the file **SetUpSite4.java** in your text editor and change the class to **SetUpSite5**.



2. Just below the declaration for `oneSite`, to provide the `SetUpSite5()` method with a variable to hold any site number returned from the `getSiteNumber()` method, type `int number;`, and then press **[Enter]**.
3. Next call the method `setSiteNumber()` to set the site number for `oneSite`. Type `oneSite.setSiteNumber(101);`. The number in parentheses could be any integer number.
4. After the statement that prints the age of the company, `System.out.println("Serving you for " + age + " years");`, to call the `getSiteNumber()` method and assign its return value to the number variable, type `number = oneSite.getSiteNumber();`, and then press **[Enter]**.
5. To add a call to the `println()` method to display the value stored in `number`, type `System.out.println("The number of the event site is " + number);`.
6. Save the program file as **SetUpSite5.java** in the Chapter.03 folder on your Student Disk.
7. Compile the program by typing `javac SetUpSite5.java`. Correct any errors and compile again, if necessary.
8. Execute the program by typing `java SetUpSite5`. Your output should look like Figure 3-26.



```
Command Prompt
A:\Chapter.03>java SetUpSite5
Event Handlers Incorporated is
dedicated to making your event
a most memorable one.
Serving you for 26 years
The number of the event site is 101
A:\Chapter.03>
```

**Figure 3-26** Output of the `SetUpSite5` program

## ORGANIZING CLASSES

Most classes you create will have more than one data field and more than two 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** Employee class with 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, the `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 common-sense rules, there is 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 salary of \$400.00, but there will be 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 `private String empLastName, empFirstName;`. However, it is usually easier to identify each Employee field at a glance if the fields are listed vertically.

Even if the only methods created for the Employee class include one set method and one get method for each instance variable, eight methods are required. Consider an Employee record for most organizations and 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 num)
    {
        empNum = num;
    }
    public String getFirstName()
    {
        return empFirstName;
    }
    public void setFirstName(String name)
    {
        empFirstName = name;
    }
    public String getLastName()
    {
        return empLastName;
    }
    public void setLastName(String name)
    {
        empLastName = name;
    }
    public double getEmpSal()
    {
        return empSalary;
    }
    public void setEmpSal(double sal)
    {
        empSalary = sal;
    }
}
```

**Figure 3-28** Employee class with data fields and 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, as shown in Figure 3-29.

```
//Programmer: Joyce Farrell
//Date April 22, 2003
//Employee.java to hold employee data

public class Employee
{
    //private data members
    private int empNum;
    private String empLastName;
    private String empFirstName;
    private double empSalary;

    //getEmpNum method returns employee number
    public int getEmpNum()
    {
        return empNum;
    }
    //setEmpNum method returns employee number
    public void setEmpNum(int num)
    {
        empNum = num;
    }
    //... and so on
}
```

**Figure 3-29** Employee class with data fields, methods, and comments



Although good program comments are crucial to creating understandable code, you will not be asked to include them in most examples in this book in an effort to save space.

**To expand the EventSite3 class to contain data fields and methods:**

1. Open the **EventSite3.java** file from the Chapter.03 folder in the text editor, and change the class name to **EventSite4**. Your program looks like Figure 3-30.

```
public class EventSite4
{
    private int siteNumber;
    public int getSiteNumber()
    {
        return siteNumber;
    }
    public void setSiteNumber(int n)
    {
        siteNumber = n;
    }
}
```

**Figure 3-30** EventSite4.java class

You will add two new data fields to the `EventSite4` class: a `double` to hold a usage fee for the site, and a `String` to hold the site manager's last name.

2. Position the insertion point at the end of the declaration of the `private int siteNumber;` variable, press **[Enter]** to start a new line, and then type `private double usageFee;` and `private String managerName;` on separate lines.

You will also enter four new methods to set and get data from each of the two new fields. To ensure that the methods are easy to locate later, you will place them in alphabetical order within the class.

3. Position the insertion point after the end of the closing curly brace of the `getSiteNumber()` method, press **[Enter]** to start a new line, and then enter the following `getUsageFee()` method:

```
public double getUsageFee()
{
    return usageFee;
}
```

4. Position the insertion point at the end of the `private String managerName;` declaration, press **[Enter]** to start a new line, and then enter the following `getManagerName()` method:

```
public String getManagerName()
{
    return managerName;
}
```

5. Position the insertion point after the closing bracket of the `getSiteNumber()` method, press **[Enter]** to start a new line, and then enter the following `setUsageFee()` method:

```
public void setUsageFee(double amt)
{
    usageFee = amt;
}
```

6. Position the insertion point after the closing curly brace of the `getUsageFee()` method, press **[Enter]**, and then enter the following `setManagerName()` method:

```
public void setManagerName(String name)
{
    managerName = name;
}
```

7. Start a line above each of the methods, and add a comment describing the function of the method.

8. Save the file as **EventSite4.java** and compile it by typing the command **javac EventSite4**. If necessary, correct any errors, save the file, and then compile again.

You have created an EventSite4 class that contains both data and methods. However, no actual event sites exist yet. You must write a program that instantiates one or more EventSite objects to give actual values to the data fields for that object, and to manipulate the data in the fields using the class methods. Next you will create a program to test the new, expanded EventSite4 class.

#### To create the test program:

1. Open a new document in the text editor, and then enter the class that tests the new expanded EventSite4 class. The class should look like Figure 3-31.

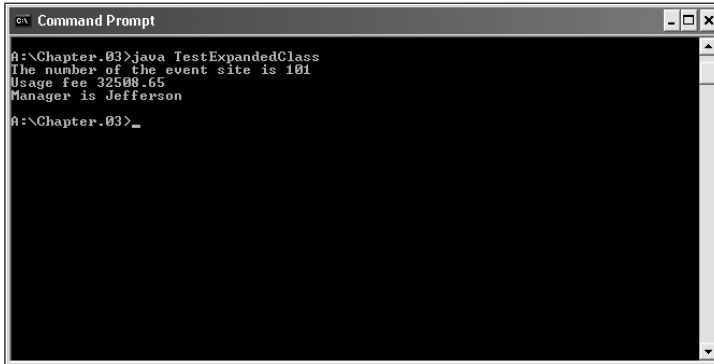
```
//Programmer: Joyce Farrell
//Date: April 22, 2003
//Program: TestexpandedClass
//Tests the expanded EventSite4 class
public class TestExpandedClass
{
    public static void main(String[] args)
    {
        EventSite4 oneSite = new EventSite4()
        oneSite.setSiteNumber(101);
        oneSite.setUsageFee(32508.65);
        oneSite.setManagerName("Jefferson");
        System.out.print("The number of the event site is ");
        System.out.println(oneSite.getSiteNumber());
        System.out.println("Usage fee "
            + oneSite.getUsageFee());
        System.out.println("Manager is "
            + oneSite.getManagerName());
    }
}
```

**Figure 3-31** The TestExpandedClass class



You should get into the habit of documenting your programs with your name, today's date, and a brief explanation of the program. Your instructor might also ask you to insert additional information as comment text.

2. Save the file as **TestExpandedClass.java** in the Chapter.03 folder on your Student Disk. Compile the program and correct any errors, if necessary.
3. Execute the class with the command-line statement **java TestExpandedClass**. Your output should look like Figure 3-32.



```
Command Prompt
A:\Chapter.03>java TestExpandedClass
The number of the event site is 101
Usage fee 32508.65
Manager is Jefferson
A:\Chapter.03>_
```

Figure 3-32 Output of the TestExpandedClass program

## USING CONSTRUCTORS

When you create a class, such as `Employee`, and instantiate an object with a statement such as `Employee chauffeur = new Employee();`, you are actually calling a method named `Employee()` that is provided by the Java compiler. A **constructor method** is a method that establishes an object. The constructor method named `Employee()` establishes one `Employee` with the identifier `chauffeur`, and provides the following specific initial values to the `Employee`'s data fields:

- Numeric fields are set to 0 (zero).
- Character fields are set to Unicode `'\u0000'`.
- The Boolean fields are set to `false`.
- The object type fields are set to null (or empty).

If you do not want an `Employee`'s fields to hold these default values, or if you want to perform additional tasks when you create an `Employee`, then you can write your own constructor method. Any constructor method you write must have the same name as the class it constructs, and constructor methods cannot have a return type. For example, if every `Employee` has a starting salary of \$300.00, then you could write the constructor method for the `Employee` class that appears in Figure 3-33. Any `Employee` instantiated will have a default `empSal` figure of 300.00.

```
Employee()  
{  
    empSal = 300.00;  
}
```

Figure 3-33 Employee class constructor

You can write any statement in a constructor. Although you usually have no reason to do so, you could print a message from within a constructor or perform any other task. Next you will add a constructor to the `EventSite4` class, and demonstrate that it is called when an `EventSite4` object is instantiated.

**To add a constructor to the `EventSite4` class:**

1. Open the **EventSite4.java** file in your text editor. Change the class name to **EventSite5**.
2. Place the insertion point at the end of the line containing the last field declaration `private String managerName;`, and then press **[Enter]** to start a new line.
3. Add the following constructor function that sets any `EventSite` `siteNumber` to 999 and any manager's name to "ZZZ" upon construction:

```
EventSite5()  
{  
    siteNumber = 999;  
    managerName = "ZZZ";  
}
```

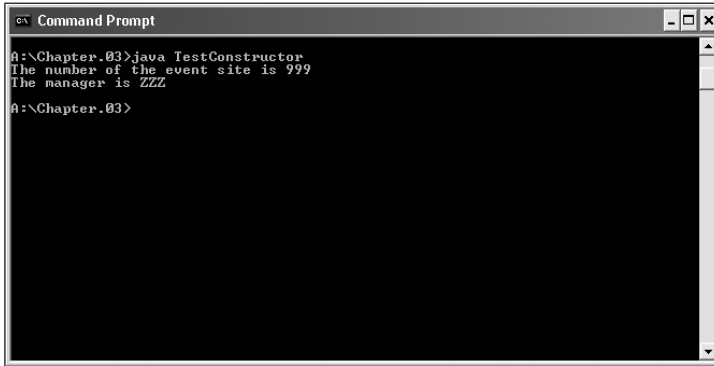
4. Save the file as **EventSite5.java**, compile it, and correct any errors.
5. Open a new text file and create a test class named **TestConstructor** using the code shown in Figure 3-34.

```
public class TestConstructor  
{  
    public static void main(String[] args)  
    {  
        EventSite5 oneSite = new EventSite5();  
        System.out.print("The number of the event site is ");  
        System.out.println(oneSite.getSiteNumber());  
        System.out.print("The manager is ");  
        System.out.println(oneSite.getManagerName());  
    }  
}
```

**Figure 3-34** TestConstructor class

6. Save the file as **TestConstructor.java** in the Chapter.03 folder on your Student Disk, compile the file, and correct any syntax errors.
7. Execute the program and confirm that it declares a `oneSite` object of type `EventSite5`, calls the constructor, and assigns the indicated initial values, as shown in Figure 3-35.





```
Command Prompt
A:\Chapter.03>java TestConstructor
The number of the event site is 999
The manager is ZZZ
A:\Chapter.03>
```

Figure 3-35 Output of the TestConstructor program

## CHAPTER SUMMARY

- A method is a series of statements that carry out a task. 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 modifiers, the return type for the method, the method name, an opening parenthesis, an optional list of method arguments, and a closing parenthesis.
- You place a method within the program that will use it, but not within any other method. If you place a method call within a class that does not contain the method, you must use the class name, followed by a dot, followed by the method. Some methods require a message or argument.
- When you write the method declaration for a method that can receive an argument, you need to include the type of the argument and a local name for the argument within the method declaration parentheses. You can call a method within a program using either a constant value or a variable as an argument.
- You can pass multiple arguments to methods by listing the arguments and separating them 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 program that calls a method.
- Objects inherit attributes from classes. Class objects have attributes and methods associated with them. Class instance methods that will be used with objects are not usually static. You can send messages to objects. Additionally, some methods used in a program must return a message or value.

- A class header contains an optional access modifier, the keyword `class`, and any legal identifier you choose for the name of your class. The instance variables, or fields, of a class are placed as statements within the class's curly braces.
- Declaring a class does not create any actual objects; you must instantiate any objects that are members of a class. 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.
- A constructor method establishes an object and provides specific initial values for the object's data fields. A constructor method 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`.

---

## REVIEW QUESTIONS

1. 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 access modifiers
  - c. arguments
  - d. parentheses
3. A public 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 method named `displayFacts()` if the method receives an `int` argument?
  - a. `public void int displayFacts()`
  - b. `public void displayFacts(int)`
  - c. `public void displayFacts(int data)`
  - d. `public void displayFacts()`

5. The method with the declaration `public int aMethod(double d)` is a method type \_\_\_\_\_.
  - a. `int`
  - b. `double`
  - c. `void`
  - d. You cannot determine the method type.
6. Which of the following is a correct call to a method declared as `double aMethod(char code)`?
  - a. `double aMethod();`
  - b. `double aMethod('V');`
  - c. `aMethod(int 'M');`
  - d. `aMethod('Q');`
7. A method is declared as `public 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 the above answers are correct.
8. The method with the declaration `public char procedure(double d)` has a method type of \_\_\_\_\_.
  - a. `public`
  - b. `char`
  - c. `procedure`
  - d. `double`
9. The method `public Boolean testValue(int response)` returns \_\_\_\_\_.
  - a. a Boolean value
  - b. an integer 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 int getVal(double sum)`?
  - a. `return;`
  - b. `return 77;`
  - c. `return 2.3;`
  - d. Any of the above could be the last coded line of the method.

11. The 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. Class objects have both attributes and \_\_\_\_\_.
  - a. fields
  - b. data
  - c. methods
  - d. instances
13. You send messages to an object through its \_\_\_\_\_.
  - a. fields
  - b. methods
  - c. classes
  - d. data
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 written \_\_\_\_\_.
  - a. as a single statement
  - b. within parentheses
  - c. between curly braces
  - d. as a method call
16. Most class 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 \_\_\_\_\_.
  - a. structured logic
  - b. object orientation
  - c. information hiding
  - d. data masking
18. When you declare a variable, as in `double salary;`, you \_\_\_\_\_.
  - a. also must explicitly allocate memory for it
  - b. need not explicitly allocate memory for it
  - c. must explicitly allocate memory for it only if it is stored in a class
  - d. can declare it to use no memory
19. If a class is named Student, then the class constructor name is \_\_\_\_\_.
  - a. any legal Java identifier
  - b. any legal Java identifier that begins with S
  - c. StudentConstructor
  - d. Student
20. If you use the default constructor, \_\_\_\_\_.
  - a. numeric fields are set to 0 (zero)
  - b. character fields are set to blank
  - c. Boolean fields are set to true
  - d. object type fields are set to 0 (zero)

---

## EXERCISES

1. Name any device you use every day. Discuss how implementation hiding is demonstrated in the way this device works. Is it a benefit or a drawback to you that implementation hiding exists for methods associated with this object?
2.
  - a. Create a class named Numbers whose `main()` method holds two integer variables. Assign values to the variables. Create two additional methods, `sum()` and `difference()`, that compute the sum of and difference between the values of the two variables, respectively. Each method should perform the computation and display the results. In turn, call each of the two methods from `main()`, passing the values of the two integer variables. Save the program as **Numbers.java** in the Chapter.03 folder on your Student Disk.
  - b. Add a method named `product()` to the Numbers class. The `product()` method should compute the multiplication product of two integers, but not display the answer. Instead, it should return the answer to the calling `main()` program, which displays the answer. Save the program as **Numbers2.java** in the Chapter.03 folder on your Student Disk.

3. Create a class named Eggs. Its main() method holds an integer variable named numberOfEggs to which you will assign a value. Create a method to which you pass numberOfEggs. The method displays the eggs in dozens; for example, 50 eggs is 4 full dozen (with 2 eggs remaining). Save the program as **Eggs.java** in the Chapter.03 folder on your Student Disk.
4. Create a class named Monogram. Its main() method holds three character variables that hold your first, middle, and last initials, respectively. Create a method to which you pass the three initials and which displays the initials twice—once in the order first, middle, last, and a second time in traditional monogram style (first, last, middle). Save the program as **Monogram.java** in the Chapter.03 folder on your Student Disk.
5. Create a class named Exponent. Its main() method holds an integer value, and in turn passes the value to a method that squares the number and to a method that cubes the number. The main() method prints the results. Create the two methods that respectively square and cube an integer that is passed to them, returning the calculated value. Save the program as **Exponent.java** in the Chapter.03 folder on your Student Disk.
6. Create a class named Cube that displays the result of cubing a number. Pass a number to a method that cubes a number and returns the result. The display should execute within the main() method that calls the cube method. Save the program as **Cube.java** in the Chapter.03 folder on your Student Disk.
7. Create a program that displays the result of a sales transaction. The calculation requires three numbers. The first number represents the product price. The second number is the salesperson commission. These two numbers should be added together. The third value represents a customer discount; subtract this third number from the result of the addition. Create two classes. The first class, Transaction, contains the method to do the calculation. The three numbers are passed to this method by a statement in the other class. The display is performed in the class that calls the calculation method. Save the program as **Calculator.java** in the Chapter.03 folder on your Student Disk.
8. Write a program that displays the result of dividing two numbers and also displays any remainder. Do the calculation and display in the same method, which is a separate method from the main() method. Save the program as **Divide.java** in the Chapter.03 folder on your Student Disk.
9.
  - a. Create a class named Pizza. Data fields include a String for toppings (such as pepperoni), an integer for diameter in inches (such as 12), and a double for price (such as 13.99). Include methods to get and set values for each of these fields. Save the class as **Pizza.java** in the Chapter.03 folder on your Student Disk.
  - b. Create a class named TestPizza that instantiates one Pizza object and demonstrates the use of the Pizza set and get methods. Save this class as **TestPizza.java** in the Chapter.03 folder of your Student Disk.

10. 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** in the Chapter.03 folder on your Student Disk.  
b. Write a class named **ShowStudent** that instantiates a **Student** object from the class you created. Compute the **Student** grade point average, and then display all the values associated with the **Student**. Save the program as **ShowStudent.java** in the Chapter.03 folder on your Student Disk.  
c. Create a constructor method for the **Student** class you created. The constructor should initialize each **Student**'s ID number to 9999 and his or her grade point average to 4.0. Write a program that demonstrates that the constructor works by instantiating an object and displaying the initial values. Save the program as **Student2.java**.
11. a. Create a class named **Circle** with fields named radius, area, and diameter. Include a constructor that sets the radius to 1. Also include methods named **setRadius()**, **getRadius()**, **computeDiameter()**, which computes a circle's diameter, and **computeArea()**, which computes a circle's area. (The diameter of a circle is twice its radius, and the area is 3.14 multiplied by the square of the radius.) Save the class as **Circle.java** in the Chapter.03 folder of your Student Disk.  
b. Create a class named **TestCircle** whose **main()** method declares three **Circle** objects. Using the **setRadius()** method, assign one **Circle** a small radius value and assign another a larger radius value. Do not assign a value to the radius of the third circle; instead, retain the value assigned at construction. Call **computeDiameter()** and **computeArea()** for each circle and display the results. Save the program as **TestCircle.java** in the Chapter.03 folder on your Student Disk.
12. a. Create a class named **Checkup** with fields that hold a patient number, two blood pressure figures (systolic and diastolic), and two cholesterol figures (LDL and HDL). Include methods to get and set each of the fields. Include a method named **computeRatio()** that divides LDL cholesterol by HDL cholesterol and displays the result. Include an additional method named **ExplainRatio()** that explains that HDL is known as "good cholesterol" and that a ratio of 3.5 or lower is considered optimum. Save the class as **Checkup.java** in the Chapter.03 folder of your Student Disk.  
b. Create a class named **TestCheckup** whose **main()** method declares four **Checkup** objects. Provide values for each field for each patient. Then display the values. Blood pressure numbers are usually displayed with a slash between the systolic and diastolic values. (Typical numbers are values such as 110/78 or 130/90.) With the cholesterol figures, display the explanation of the cholesterol ratio calculation. (Typical numbers are values such as 100 and 40 or 180 and 70.) Save the program as **TestCheckup.java** in the Chapter.03 folder on your Student Disk.

13. Write a program that displays employee IDs and first and last names of employees. Use two classes. The first class named `Emp` contains the employee data and separate methods to set the IDs and names. The other class creates objects for the employees and uses the objects to call the set methods. Create several employees and display their data. Save the program as **Employee.java** in the Chapter.03 folder on your Student Disk.
14. Write a program that displays an invoice of several items. It should contain the item name, quantity, price, and total cost on each line for the quantity and item cost. Use two classes. The first class `Inv` contains the item data and methods to get and set the item name, quantity, and price. The other class creates objects for the items and uses the objects to call the set and get methods. Save the program as **Invoice.java** in the Chapter.03 folder on your Student Disk.
15. Write a program that schedules several meetings for a meeting room. It should contain the day of the week, starting time, and ending time for each meeting. Use two classes. The first class contains the meeting data and methods to get and set the day of the week and starting and ending times. The other class creates objects for the meetings and uses the objects to call the set and get methods. Save both classes in the program as **RoomSchedule.java** in the Chapter.03 folder on your Student Disk.
16. Write a program that calculates and displays the weekly salary for an employee who earns \$25 an hour, works 40 regular hours, 13 overtime hours, and earns time and one-half ( $\text{wage} * 1.5$ ) for overtime hours worked. Create a separate method to do the calculation and return the result to be displayed. Save the program as **Salary.java** in the Chapter.03 folder on your Student Disk.
17.
  - a. Write a program that calculates and displays the conversion of \$57 into dollar-bill form—20's, 10's, 5's, and 1's. Create a separate method to do the calculation and display. Pass 57 as a variable to this method. Save the program as **Dollars.java** in the Chapter.03 folder on your Student Disk.
  - b. In the `Dollars.java` program, alter the value of the variable that holds the amount of money. Run the program and confirm that the amount of each denomination calculates correctly.
18. Write a program that calculates and displays the amount of money you would have if you invested \$1,000 at 5 percent interest for one year. Create a separate method to do the calculation and return the result to be displayed. Save the program as **Interest.java** in the Chapter.03 folder on your Student Disk.
19.
  - a. Create a bank account named `Account`. The class should have one instance variable named `balance`. Write two constructors, one to set the value of `balance` to 0.0 when called, and a second that will receive a `balance` as a double value passed to the constructor. Write instance methods to add to, subtract from, and set the `balance` to 0.0. Write other instance methods as needed.
  - b. Write a program to instantiate an `Account` object. With the `Account` object, open an account, add a deposit to the account, withdraw an amount from the



account, and close the account. After each transaction, print the Account balance. Save the program as **TestAccount.java** in the Chapter.03 folder on your Student Disk.

20. Each of the following files saved in the Chapter.03 folder on your Student Disk has syntax and/or logical 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.
- DebugThree1.java
  - DebugThree2.java
  - DebugThree3.java
  - DebugThree4.java

## CASE PROJECT



Event Handlers wants to develop an Employee program to set and retrieve employee ID numbers, employee salaries, and first and last employee names. The current list of employees and their data are as follows:

**Table 3-1** Employee data

Employee Name	Employee ID	Employee Salary
Kim Yee	101	\$40,000.00
John Reynolds	102	\$55,000.00
Elena Gonzales	103	\$50,500.00
Jim O'Shea	104	\$75,000.00

The program will require a constructor that sets the names to “Unknown”, ID’s to “0”, and Salaries to “0.00”. As a programmer, your task is to write the necessary classes to accomplish the tasks given by the specifications.

