

# ADVANCED OBJECT CONCEPTS

**In this chapter, you will:**

- ◆ Understand blocks and scope
- ◆ Overload a method
- ◆ Learn about ambiguity
- ◆ Send arguments to constructors
- ◆ Overload constructors
- ◆ Learn about the `this` reference
- ◆ Work with constants
- ◆ Use automatically imported, prewritten constants and methods
- ◆ Use prewritten imported methods
- ◆ Learn about Gregorian calendars

Lynn Greenbrier, your mentor at Event Handlers Incorporated, pops her head into your cubicle on Monday morning. “How’s the programming going?” she asks.

“I’m getting the hang of using objects,” you tell her, “but I want to create lots of objects, and it seems like I will need so many methods for the classes that I create that it will be very hard to keep track of them.” You pause a moment and add, “And all these set methods are driving me crazy. I wish an object could just start with values.”

“Anything else bothering you?” Lynn asks.

“Well,” you reply, “since you asked, shouldn’t some objects and methods that are used by all kinds of programmers already be created for me? I can’t be the first person who ever thought about taking a square root of a number or calculating a billing date for 10 days after service.”

“You’re in luck!” Lynn smiles. “Java’s creators have already thought about these things. Let me tell you about some of the more advanced things you can do with your classes.”

## UNDERSTANDING BLOCKS AND SCOPE

Within any class or method, the code between a pair of curly braces is called a **block**. For example, the program shown in Figure 4-1 contains two blocks. The first block, or **outside block**, begins immediately after the method declaration and ends at the end of the method. The second block, or **inside block**, is contained within the second set of curly braces and contains three statements: the declaration of anotherNumber and two println() statements. The inside block is **nested** within the outside block. A block can exist entirely within another block, or entirely outside and separate from another block, but blocks can never overlap.

```
public static void methodWithTwoBlocks()
{
    int aNumber = 22;
    // aNumber comes into existence
    System.out.println("Number is " + aNumber);
    {
        int anotherNumber = 99;
        // anotherNumber comes into existence
        System.out.println("aNumber is " + aNumber);
        System.out.println("anotherNumber is " +
            anotherNumber);
    } // End of block - anotherNumber ceases to exist
    System.out.println("aNumber is " + aNumber);
} // End of outer block - aNumber ceases to exist
```

**Figure 4-1** The methodWithTwoBlock() method

If you declare a variable in one program that you write, you cannot use that variable in another program. Similarly, when you declare a variable within a block, you cannot reference that variable outside the block. The portion of a program within which you can reference a variable is the variable's **scope**. A variable comes into existence, or **comes into scope**, when you declare it. A variable ceases to exist, or **goes out of scope**, at the end of the block in which it is declared.



Although you can create as many variables and blocks as you need within any program, it is not wise to do so without a reason. The use of unnecessary variables and blocks increases the likelihood of improper use of variable names and scope.

In the methodWithTwoBlocks() method shown in Figure 4-1, the variable aNumber exists from the point of its declaration until the end of the method. This means aNumber exists both in the outer block and within the inner block, and can be used anywhere in the method. The variable anotherNumber comes into existence within the inner block; anotherNumber ceases to exist when the inner block ends, and cannot be used beyond its block.

Figure 4-2 shows some invalid statements. The first assignment `aNumber = 75;` is invalid because `aNumber` has not been declared yet. Similarly, Invalid statement 2, `anotherNumber = 489;`, is invalid because it has not been declared yet. Statement 3 is also invalid because `anotherNumber` still has not been declared. After you declare `anotherNumber`, you can use it for the remainder of the block, but Invalid statement 4 is outside the block and `anotherNumber` has gone out of scope. The last statement in Figure 4-2, `aNumber = 29;`, will not work because it falls outside the block in which `aNumber` was declared; it actually falls outside the `methodWithTwoBlocks()` method.

```
public static void methodWithTwoBlocks()
{
    aNumber = 75; // Invalid statement
    int aNumber = 22;
    System.out.println("aNumber is " + aNumber);
    anotherNumber = 489; // Invalid statement 2
    {
        anotherNumber = 165; // Invalid statement 3
        int anotherNumber = 99;
        System.out.println("aNumber is " + aNumber);
        System.out.println("anotherNumber is " +
            anotherNumber);
    }
    System.out.println("aNumber is " + aNumber);
    System.out.println("anotherNumber is " +
        anotherNumber); // Invalid statement 4
}
anumber = 29; // Invalid statement 5
```

**Figure 4-2** The `methodWithTwoBlock()` method with some invalid statements



You are not required to vertically align the opening and closing braces for a block, but your programs are much easier to read if you do.

Within a method, you can declare a variable with the same name multiple times, as long as each declaration is in its own, nonoverlapping block. For example, the two declarations of variables named `someVar` in Figure 4-3 are valid because each variable is contained within its own block. The first instance of `someVar` has gone out of scope before the second instance comes into scope.

```

public static twoDeclarations()
{
    { // Begin first block
        int someVar = 7;
        System.out.println(someVar);
    } // End first block
    { // Begin second block
        int someVar = 845;
        System.out.println(someVar);
    } // End second block
}

```

**Figure 4-3** The twoDeclarations() method

You cannot declare the same variable name more than once within a block. For example, in Figure 4-4, the second declaration of `aValue` causes an error because you cannot declare the same variable twice within the outer block of the method. By the same reasoning, the third declaration of `aValue` is also invalid, even though it appears within a new block. The block that contains the third declaration is entirely within the outside block, so the first declaration of `aValue` has not gone out of scope.

```

public static methodWithRedeclarations()
{
    int aValue = 35;
    System.out.println(aValue);
    int aValue = 99; // Invalid - second declaration
    {
        int anotherValue = 58; // Valid
        int aValue = 99; // Invalid - third declaration
        // This block is inside the outer block
    }
}

```

**Figure 4-4** Invalid methodWithRedeclarations()

If you declare a variable within a class, and use the same variable name within a method of the class, then the variable used inside the method takes precedence, or **overrides**, the first variable. For example, consider a class that holds Employee information including two integer fields, `aNum` and `aDept`, as shown in Figure 4-5.

Figure 4-5 shows an `Employee` class with two integers and two void methods. When a `TestEmployee2` program instantiates an `Employee` object with a statement such as `Employee adminAssistant = new Employee();`, then either `empMethod()` or `anotherEmpMethod()` can be called using the `adminAssistant` object and the dot operator (`.`), as in `adminAssistant.empMethod()` or `adminAssistant.anotherEmpMethod()`.

```
public class Employee
{
    private int aNum = 44;
    private int aDept = 55;
    public void empMethod()
    {
        int aNum = 88;
        //aNum overrides the class variable aNum
        System.out.println("aNum is " + aNum;
        System.out.println("aDept is " + aDept;
    }
    public void anotherEmpMethod()
    {
        System.out.println("aNum is " + aNum;
        System.out.println("aDept is " + aDept;
    }
}
```

**Figure 4-5** Employee2 class with an overriding variable

When the method call is `adminAssistant.empMethod()`, the output will indicate that `aNum` is 88 and `aDept` is 55. The `empMethod()` will use the local `aNum` valued at 88, but use the class `aDept` valued at 55. When the method call is `adminAssistant.anotherEmpMethod()`, the output will show that `aNum` is 44 and `aDept` is 55; in both cases, the class variables are used because they have not been overridden within `anotherEmpMethod()`. When you write programs, it is best to avoid confusing situations that arise when you give the same name to a class variable and a method variable. But, if you do use the same name, be aware that within the method, the method variable will override the class variable.

Next you will create a method with several blocks to demonstrate block scope.

**To demonstrate block scope:**

1. Start your text editor, and then open a new document, if necessary.
2. Type the header for a class named `DemoBlock` as **public class DemoBlock**. On the next three lines, type the opening curly brace (`{`), the `main()` method header, **public static void main(String[] args)**, and the `main()`'s opening curly brace (`{`).
3. On a new line that is indented one column, declare an integer by typing:  
**int x = 1111;**
4. On new, indented lines, type the following two `println()` statements:  
**System.out.println("Demonstrating block scope");**  
**System.out.println("In first block x is " + x);**

5. Begin a new block by typing an opening curly brace on the next line. Within the new block, declare another integer by typing `int y = 2222;`. Within this new block, type the following two statements to display the values of x and y:

```
System.out.println("In second block x is " + x);
System.out.println("In second block y is " + y);
```

6. End the block by typing a closing curly brace (`}`). On the next line, begin a new block with an opening curly brace. Within this new block, declare a new integer with the same name as the integer declared in the previous block by typing `int y = 3333;`
7. Enter two `println()` statements, a method call, and two more `println()` statements, as follows:

```
System.out.println("In third block x is " + x);
System.out.println("In third block y is " + y);
demoMethod();
System.out.println("After method x is " + x);
System.out.println("After method block y is " + y);
```

8. Close this block by typing a closing curly brace.
9. Type `System.out.println("At the end x is " + x);`, and then type a closing curly brace. This last statement in the program displays the value of x.
10. Finally, enter the following `demoMethod()` that creates its own x and y, assigns different values, and then displays them:

```
public static void demoMethod()
{
    int x = 8888, y = 9999;
    System.out.println("In demoMethod x is " + x);
    System.out.println("In demoMethod block y is " + y);
}
```

11. Type the final closing curly brace, and then save the file as **DemoBlock.java** in the Chapter.04 folder on your Student Disk. At the command prompt, compile the file by typing the command `javac DemoBlock.java`. If necessary, correct any errors and compile again.
12. Run the program by typing the command `java DemoBlock`. Your output should look like Figure 4-6.

It is important to understand the impact that blocks have on your variables. Once you understand the scope of variables, you can more easily locate the source of many errors within your programs.

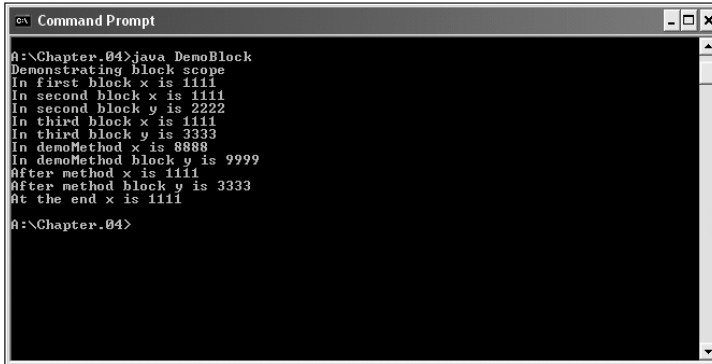


Figure 4-6 Output of the DemoBlock program



To gain a more complete understanding of blocks and scope, change the values of *x* and *y* in the program, and try to predict the exact output before resaving, recompiling, and rerunning the program.

## OVERLOADING A METHOD

**Overloading** involves using one term to indicate diverse meanings, or writing multiple methods with the same name, but with different arguments. When you use the English language, you overload words all the time. When you say “open the door,” “open your eyes,” and “open a computer file,” you are talking about three very different actions using very different methods, and producing very different results. However, anyone who speaks English fluently has no trouble understanding your meaning because the verb *open* is understood in the context of the noun that follows it.

When you overload a Java method, you write multiple methods with a shared name. The compiler understands your meaning based on the arguments you use with the method. For example, suppose you create a class method to apply a simple interest rate to a bank balance. The method receives two double arguments—the balance and the interest rate—and displays the multiplied result. Figure 4-7 shows the method.

```
public static void simpleInterest(double bal, double rate)
{
    double interest;
    interest = bal * rate;
    System.out.println("Interest on " + bal + " at " +
        rate + " interest rate is " + interest);
}
```

Figure 4-7 The `simpleInterest()` method with two double arguments



The `simpleInterest()` method can receive integer arguments even though it is defined as needing double arguments because integers will be promoted or cast automatically to doubles, as you learned in Chapter 1.

When a program calls the `simpleInterest()` method and passes double values, as in `simpleInterest(1000.00, 0.04)`, the simple interest will be calculated correctly as four percent of \$1000.00. Assume, however, that the interest rate passed to the `simpleInterest()` method comes from inconsistent user input. Some users who want to indicate an interest rate of four percent might type `.04`; others might type `4` and assume that they are typing four percent. When the `simpleInterest()` method is called with the arguments \$1000.00 and `.04`, the interest is calculated correctly as 40.00. When the method is called using \$1000.00 and `4`, the interest is calculated incorrectly as 4000.00.

A solution for the conflicting use of numbers to represent parameter values is to overload the `simpleInterest()` method. For example, in addition to the `simpleInterest()` method shown in Figure 4-7, you could add the method shown in Figure 4-8.

```
public static void simpleInterest(double bal, int rate)
// Notice rate type
{
    double interest, rateAsPercent;
    rateAsPercent = rate/100.0;
    // Converts whole number rate to decimal equivalent
    interest = bal * rateAsPercent;
    System.out.println("Interest on " + bal + " at " +
        rate + " interest rate is " + interest);
}
```

**Figure 4-8** The `simpleInterest()` method with a double and an integer argument



Note the `rateAsPercent` figure is calculated by dividing by `100.0`, and not by `100`. If two integers are divided, the result is a truncated integer; dividing by a double `100.0` causes the result to be a double. Alternately, you could use an explicit cast such as `(double)rate/100.00`.

If the method `simpleInterest()` is called using two double arguments, as in `simpleInterest(1000.00, .04)`, the first `simpleInterest()` method shown in Figure 4-7 will execute. However, if an integer is used as the second parameter in the call to `simpleInterest()`, as in `simpleInterest(1000.00, 4)`, then the method shown in Figure 4-8 will execute. The whole number rate figure will be divided by `100.0` correctly before it is used to determine the interest earned.



Of course, you could use methods with different names to solve the dilemma of producing an accurate simple interest figure—for example, `simpleInterestRateUsingDouble()` and `simpleInterestRateUsingInt()`. Using this approach requires that you place a decision within your program to determine which of the two methods to call, but it is more convenient to use one method name and then let the interpreter determine which method to use. Also, it is easier to remember one reasonable name for tasks that are functionally identical except for argument types.



You will learn about placing a decision within your program in Chapter 5.

Next you will overload methods to display event dates for Event Handlers Incorporated. The methods will take one, two, or three integer arguments. If there is one argument, it is the month, and the event is scheduled for the first day of the given month in the year 2003. If there are two arguments, they are the month and the day in the year 2003. Three arguments represent the month, day, and year.



In addition to creating your own class to store dates, you can use a built-in Java class to handle dates. You will learn about this class later in this chapter.

### To overload an `overloadDate()` method to take one, two, or three arguments:

1. Open a new file in your text editor.
2. Create the following `DemoOverload` class, with three integer variables and three calls to an `overloadDate()` method:

```
public class DemoOverload
{
    public static void main(String[] args)
    {
        int month = 6, day = 24, year = 2003;
        overloadDate(month);
        overloadDate(month, day);
        overloadDate(month, day, year);
    }
}
```

3. Create the following `overloadDate()` method that requires one argument:

```
public static void overloadDate(int mm)
{
    System.out.println("Event date " + mm + "/1/2003");
}
```

4. Create the following overloadDate() method that requires two arguments:

```
public static void overloadDate(int mm, int dd)
{
    System.out.println("Event date " + mm + "/" +
        dd + "/2003");
}
```

5. Create the following overloadDate() method that requires three arguments:

```
public static void overloadDate(int mm, int dd, int yy)
{
    System.out.println("Event date " + mm + "/" +
        dd + "/" + yy);
}
```

6. Type the closing curly brace for the DemoOverload class.
7. Save the file as **DemoOverload.java** in the Chapter.04 folder on your Student Disk.
8. Compile the program, correct any errors, recompile if necessary, and then execute the program. Figure 4-9 shows the output. Notice that whether you call the overloadDate() method using one, two, or three arguments, the date prints correctly because you have successfully overloaded the overloadDate() method.

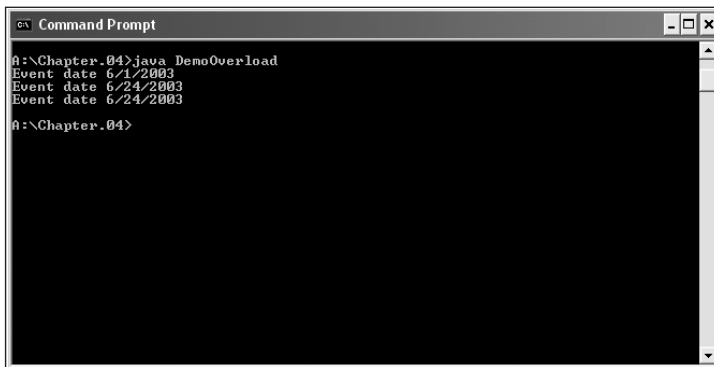


Figure 4-9 Output of the DemoOverload program

---

## LEARNING ABOUT AMBIGUITY

When you overload a method, you run the risk of creating an **ambiguous** situation—one in which the compiler cannot determine which method to use. For example, consider the simple method shown in Figure 4-10.

```
void simpMeth(double d)
{
    System.out.println("Method receives double parameter");
}
```

**Figure 4-10** The simpMeth() method with a double argument

If you declare `doubleValue` as a double variable, and `intValue` as an int variable, then either method call `simpMeth(doubleValue);` or `simpMeth(intValue);` results in the output “Method receives double parameter”. When you call the method with the double argument, the method works as expected. When you call the method with the integer argument, then the integer is cast as (or promoted to) a double, and the method also works.



Note that if the method with the declaration `void simpMeth(double d)` did not exist, but the declaration `void simpMeth(int i)` did exist, then the method call `simpMeth(doubleValue);` would fail. Although an integer can be promoted to a double, a double cannot become an integer. This makes sense if you consider the potential loss of information when a double value is reduced to an integer.

If you add a second overloaded `simpMeth()` method within a program that takes an integer parameter (as shown in Figure 4-11), then the output changes when you call `simpMeth(intValue);`. Instead of promoting an integer argument to a double, the compiler recognizes a more exact match for the method call that uses the integer argument, so it calls the version of the method that produces the output “Method receives integer parameter”.

```
void simpMeth(int i)
{
    System.out.println("Method receives integer parameter");
}
```

**Figure 4-11** The simpMeth() method with an integer argument

A more complicated and potentially ambiguous situation arises when the compiler cannot determine which of several versions of a method to use. Consider the following overloaded `simpleInterest()` method declarations:

```
public static void simpleInterest(double bal, double rate)
public static void simpleInterest(double bal, int rate)
// Notice rate type
```

A call to `simpleInterest()` with two double arguments executes the first version of the method, and a call to `simpleInterest()` with a double and an integer argument executes the second version of the method. With each of these calls, the compiler can find an

exact match for the arguments you send. However, if you call `simpleInterest()` using two integer arguments, as in `simpleInterest(300,6);`, an ambiguous situation arises because there is no exact match for the method call. Because two integers can be promoted to two doubles (thus matching the first version of the overloaded method), or just one integer can be promoted to a double (thus matching the second version), the compiler does not know which version of the `simpleInterest()` method to use and the program will not execute. You could argue that *int, int* is “closer” to *double, int* than it is to *double, double*, but the compiler does not make such decisions for you.



An overloaded method is not ambiguous on its own—it only becomes ambiguous if you create an ambiguous situation. A program containing a potentially ambiguous situation will run problem free if you do not make any ambiguous method calls.

It is important to note that you can overload methods correctly by providing different argument lists for methods with the same name. Methods with identical names that have identical argument lists, but different return types, are not overloaded—they are illegal. For example, `int aMethod(int x)` and `void aMethod(int x)` cannot coexist within a program. The compiler determines which of several versions of a method to call based on argument lists. When the method call `aMethod(17);` is made, the compiler will not know which method to execute because both methods take an integer argument.

---

## SENDING ARGUMENTS TO CONSTRUCTORS

In Chapter 3, you learned that Java automatically provides a constructor method when you create a class. You also learned that you can write your own constructor method, and that you often do so when you want to ensure that fields within classes are initialized to some appropriate default value. Additionally, you can write constructor methods that receive arguments. Such arguments are often used for initialization purposes when the values that you want to assign to objects upon creation might vary.

For example, consider the `Employee` class with two data fields shown in Figure 4-12. Its constructor method assigns 999 to each potentially instantiated `Employee`'s `empNum`. Any time an `Employee` object is created using a statement such as `Employee partTimeWorker = new Employee();`, even if no other data-assigning methods are ever used, you are ensured that the `partTimeWorker` `Employee`, like all `Employees`, will have an initial `empNum` of 999.

```
public class Employee
{
    private int empNum;
    private double empSalary;
    // Constructor method
    Employee()
    {
        empNum = 999
    }
    // Other methods go here
}
```

**Figure 4-12** Employee class



You can use a `setEmpNum()` method to assign values to individual Employee objects after construction, but a constructor method assigns the values at the time of creation.

Alternately, you might choose to create Employees with initial `empNums` that differ for each Employee. To accomplish this within a constructor, you need to pass an employee number to the constructor. Figure 4-13 shows an Employee constructor that receives an argument. With this constructor, an argument is passed using a statement, such as `Employee partTimeWorker = new Employee(881);`. When the constructor executes, the integer within the method call is passed to `Employee()` and assigned to the `empNum`.

```
Employee(int num)
{
    empNum = num;
}
```

**Figure 4-13** Employee constructor method with an argument

To demonstrate a constructor with an argument, you will use a new commented version of the `EventSite5` class you created in Chapter 3.

**To alter a constructor:**

1. Open a new file in your text editor, and then enter the `EventSite6` class shown in Figure 4-14. This file is similar to the `EventSite5.java` text file you created in Chapter 3, but comments have been added for clarity. Save the file as **EventSite6.java** in the Chapter.04 folder on your Student Disk.

```

public class EventSite6
{
    private int siteNumber;
    private double usageFee;
    private String managerName;
    //Constructor
    EventSite
    {
        siteNumber = 999;
        managerName = "ZZZ";
    }
    //getManagerName() gets managerName
    public String getManagerName()
    {
        return managerName;
    }
    //getSiteNumber() gets the siteNumber
    public int getSiteNumber()
    {
        return siteNumber;
    }
    //getUsageFee() gets the usageFee
    public double getUsageFee()
    {
        return usageFee;
    }
    //setManagerName() assigns a name to the manager
    public void setManagerName(String name)
    {
        managerName = name;
    }
    //setSiteNumber() assigns a siteNumber
    public void setSiteNumber(int n)
    {
        siteNumber = n;
    }
    //setUsageFee() assigns a value to the usage fee
    public void setUsageFee(double amt)
    {
        usageFee = amt;
    }
}

```

**Figure 4-14** EventSite6.java class

2. Modify the existing constructor by typing the following code so that the constructor takes an argument for the site number and then assigns the argument value to the siteNumber field:

```

EventSite6(int siteNum)
{
    siteNumber = siteNum;
    managerName = "ZZZ";
}

```

3. Save the file and then compile and correct any errors.
4. Open a new text file to create a short program to demonstrate the constructor at work by typing the following code:

```
public class DemoConstruct
{
    public static void main(String[] args)
    {
        EventSite6 aSite = new EventSite6(678);
        System.out.println("Site number is "
            + aSite.getSiteNumber());
    }
}
```

5. Save the file as **DemoConstruct.java** in the Chapter.04 folder, and then compile and test the program. The site number (678) should be assigned to the aSite object.

## OVERLOADING CONSTRUCTORS

If you create a class from which you instantiate objects, Java automatically provides you with a constructor. Unfortunately, if you create your own constructor, the automatically created constructor no longer exists. Therefore, once you create a constructor that takes an argument, you no longer have the option of using the automatic constructor that requires no arguments.

Fortunately, as with any other method, you can overload constructors. Overloading constructors provides you with a way to create objects with or without initial arguments, as needed. For example, in addition to using the provided constructor method shown in Figure 4-14, you can create the second constructor method for the Employee class shown in Figure 4-15. When both constructors reside within the Employee class, you have the option of creating an Employee object, either with or without an initial empNum value. When you create an Employee object with `Employee aWorker = new Employee();`, the constructor with no arguments is called and the Employee object receives an initial empNum value of 999. When you create an Employee object with `Employee anotherWorker = new Employee(7677);`, the constructor that requires an integer is used, and the anotherWorker Employee receives an initial empNum of 7677.

```
Employee()
{
    empNum = 999;
}
```

**Figure 4-15** Employee constructor method with no argument

Similarly, if you want to pass values to initialize field values for `siteNumber` and `managerName`, you can create a constructor that requires two arguments. You can use the arguments to initialize field values, but you can also use arguments for any other purpose. For example, you could use the presence or absence of an argument simply to determine which of two possible constructors to call, yet not make use of the argument within the constructor method. As long as the constructor argument lists differ, there is no ambiguity in which constructor method to call.

Next you will overload the `EventSite6` constructor to take either no arguments, in which case the site number is 999, or to take an argument that is the site number.

#### To overload the `EventSite6` constructor:

1. In your text editor, open the **EventSite6.java** text file from the Chapter.04 folder on your Student Disk, and save it as **EventSite7.java**. Change the class name to **EventSite7**.
2. Position the insertion point at the end of the comment `// Constructor`, type **s** to make the word *Constructors*, and then press **[Enter]** to start a new line.
3. Above the existing constructor that requires an argument, add the new overloaded constructor that requires no argument by typing the following:

```
EventSite7()
{
    siteNumber = 999;
    managerName = "ZZZ";
}
```

4. Rename the old `EventSite6` constructor to **EventSite7**.
5. Save the file, compile, and correct any errors.
6. In your text editor, open the **DemoConstruct.java** file from the Chapter.04 folder on your Student Disk, and change the class name to **DemoConstruct2**.
7. Change the statement `EventSite6 aSite = new EventSite6(678);`, to **EventSite7 aSite = new EventSite7(678);**, position the insertion point at the end of this statement and then press **[Enter]** to start a new line.
8. Create a new `EventSite7` with no constructor argument by typing **EventSite7 anotherSite = new EventSite7();**
9. Position the insertion point after the `println()` statement that displays the site number of `aSite`, `System.out.println("Site number is " + aSite.getSiteNumber());`, and then press **[Enter]** to start a new line. Then type the following statement to print the site number of `anotherSite`:

```
System.out.println("Another site number is "
    + anotherSite.getSiteNumber());
```



10. Save the program, compile, and test the program. The two site numbers should print as 678 and 999.
11. Close your text editor.

## LEARNING ABOUT THE `this` REFERENCE

When you start creating classes from objects that you instantiate, the classes can become large very quickly. Besides data fields, each class can have many methods, including several overloaded versions. If you instantiate many objects of a class, the computer memory requirements can become substantial. Fortunately, it is not necessary to store a separate copy of each variable and method for each instantiation of a class.

Usually you want each instantiation of a class to have its own data fields. If an `Employee` class contains fields for employee number, name, and salary, every individual `Employee` object will need a unique number, name, and salary values. However, when you create a method for the `Employee` class, any `Employee` object can use the same method. Whether the method performs a calculation, sets a field value, or constructs an object, the instructions are the same for each instantiated object. Therefore, you store just one copy of a method that all instantiated objects use.

When you use an object method, you use the object name, a dot, and the method name—for example, `aWorker.getEmpNum()`; . When you refer to the `aWorker.getEmpNum()` method, you are referring to the general, shared `Employee` class `getEmpNum()` method; `aWorker` has access to the method because `aWorker` is a member of the `Employee` class. However, within the `getEmpNum()` method, when you access the `empNum` field, you access `aWorker`'s private, individual copy of the `empNum` field. Because many `Employees` might exist, but just one copy of the method exists no matter how many `Employees` there are, when you call `aWorker.getEmpNum()`, the compiler must determine *whose* copy of `empNum` should be returned by the single `getEmpNum()` method.

The compiler accesses the correct object's field because you implicitly pass to the `getEmpNum` method a reference to `aWorker`. This reference is called the `this` reference and is a reserved word in Java. For example, the two `getEmpNum()` methods shown in Figure 4-16 perform identically. The first method simply uses the `this` reference without you being aware of it; the second method uses the `this` reference explicitly.



When you pass a reference, you pass a memory address.

Usually you neither want nor need to refer to the `this` reference within the methods you write, but the `this` reference is always there, working behind the scenes, so that the data field for the correct object can be accessed.

```

public void getEmpNum()
{
    return empNum;
}
public void getEmpNum()
{
    return this.empNum;
}

```

**Figure 4-16** The `getEmpNum()` methods with implicit and explicit `this` references



Recall that methods associated with individual objects are instance methods.

In Chapter 3, you learned that most methods you create within a class are nonstatic—methods that you associate with individual objects. You also created static methods. For example, the `main()` method in a program and the method's `main()` calls without an object reference are static. These methods do not have a `this` reference because they have no object associated with them; therefore, they are called **class methods**.

You can also create **class variables**, which are variables that are shared by every instantiation of a class. For example, you might have a company ID number that is the same for all `Employee` objects. You can add a static class variable to the class definition, as shown in Figure 4-17. Also shown in figure 4-17 is a simple method to display the employee number along with the employee's `COMPANY_ID`.

```

public class Employee
{
    static private int COMPANY_ID = 12345;
    private int empNum;
    private double empSalary;
    Employee(int num)
    // Constructor requiring employee number
    empNum = num;
    }
    public void showCompanyID()
    {
        System.out.println("Worker " + empNum
            + " has company ID " + COMPANY_ID);
    }
    // Other class methods can go here
}

```

**Figure 4-17** Employee class with a static ID number field

No matter how many `Employee` objects are eventually instantiated, each will refer to the single `COMPANY_ID` field. For example, if two `Employees` are created with `Employee firstWorker = new Employee(444);` and `Employee secondWorker = new Employee(777);`, when you write the statement `firstWorker.showCompanyID()`, its output is `Worker 444 has company ID_12345`, and when you write `secondWorker.showCompanyID();`, the statement's output is `Worker 777 has COMPANY_ID 12345`. The different workers have individual IDs, but the same company ID.

Additionally, if you change the value of `COMPANY_ID` in the `Employee` class, the value changes for all class instantiations. Therefore, besides values such as a company ID, good candidates for static class variables are fields such as a legal minimum wage or a maximum number of hours that an employee is allowed to work in a single week. When such values change for one employee, they change uniformly for all employees.

## WORKING WITH CONSTANTS

In Chapter 2, you learned to create literal constants within a program. A literal constant is a fixed value that does not change, such as the literal string “First Java program.” Variables, on the other hand, do change. When you declare `int empNum;`, you expect that the value stored in `empNum` will be different at different times or for different employees.

Sometimes, however, a variable or data field should be constant; that is, it should not be changed during the execution of a program. This is known as a **constant variable**. While the concept of a constant variable is somewhat of an oxymoron, there are situations where using a constant variable is reasonable. For example, the value for a company ID is fixed, so you do not want any methods to alter the company ID value while a program is running. To prevent alteration, insert the keyword `final` in the company ID declaration. Then the name `COMPANY_ID` becomes a **symbolic constant**, which indicates that when you compile any program that uses an object that contains the `COMPANY_ID`, the field has a `final`, unalterable value. By convention, constant fields are written using all uppercase letters. The compiler does not require using uppercase identifiers for constants, but using uppercase identifiers helps you distinguish symbolic constants from variables. For readability, you can insert underscores between words in symbolic constants.



Mathematical constants are good candidates for receiving final status. For example, when `PI` is defined as `static final double PI = 3.14159;`, it appropriately becomes a constant that should never take on any other value. A fixed sales tax rate `static final double SALES_TAX = 0.075;` remains fixed for every use within a program.



You can use the keyword `final` with methods or classes. When used in this manner, `final` indicates limitations placed on inheritance. You will learn more about inheritance as you become more proficient at object-oriented programming.

You cannot change the value of a symbolic constant after declaring it; any attempt to do so will result in a compiler error. You must initialize a constant with a value; this makes sense when you consider that a constant cannot be changed later. If a constant does not receive a value upon creation, it can never receive a value. Figure 4-18 shows a typical declaration of a constant.

```
public class Employee
{
    static final private int COMPANY_ID = 12345;
    // Rest of class goes here
}
```

**Figure 4-18** Employee class with the symbolic constant `COMPANY_ID`

A constant always has the same value within a program, so you might wonder why you cannot use the actual, literal value. For example, why not code 12345 when you need the company ID rather than going to the trouble of creating the `COMPANY_ID` symbolic constant? There are at least three good reasons to use the symbolic constant rather than the literal one:

- The number 12345 is more easily recognized as the company ID if it is associated with an identifier such as `COMPANY_ID`.
- If the company ID changes, you would change the value of `COMPANY_ID` at one location within your program—where the constant is defined—rather than searching for every use of 12345 to change it to a different number. Also, being able to make the change at one location saves you valuable programming time.
- Even if you are willing to search for every instance of 12345 in a program to change it to the new company ID value, you might inadvertently change the value to one that is being used for something else, such as an employee's employee number or salary.

Next you will create a class variable to hold the location of the company headquarters for Event Handlers Incorporated. The location of the company headquarters is an ideal candidate for a class variable. Because the headquarters location is the same for every event no matter where the actual event is held, the value for the headquarters location should be stored just once, but every `EventSite7` object should have access to the information.

**To create a class variable for the EventSite7 class:**

1. In your text editor, open the **EventSite7.java** text file from the Chapter.04 folder on your Student Disk, and then change the class name to **EventSite8**. Save the file as **EventSite8.java**.
2. Position the insertion point after the opening curly brace of the class, and then press **[Enter]** to start a new line.
3. Type the class variable:

```
static final public String HEADQUARTERS = "Crystal Lake,  
IL";
```



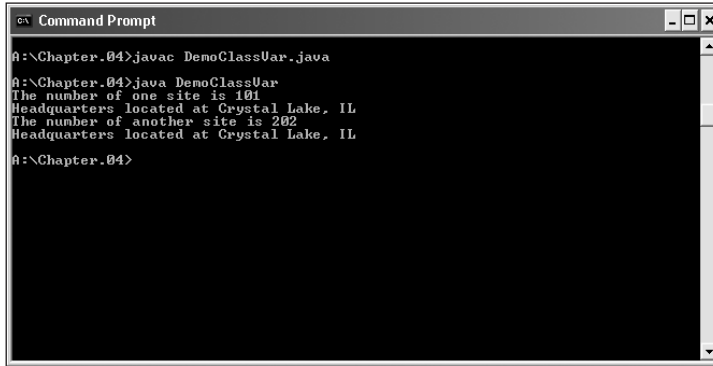
A static variable can be either **public** or **private**. If the variable is **private**, then you must write a method in your class to access it.

4. Change the **EventSite7()** constructor to **EventSite8()**, and then change the **EventSite7(int siteNum)** constructor to **EventSite8(int siteNum)**.
5. Save the file and compile.
6. Start a new file in your text editor, and then create the demonstration program named **DemoClassVar** shown in Figure 4-19. This program shows the headquarters location is the same for all EventSites.

```
public class DemoClassVar  
{  
    public static void main(String[] args)  
    {  
        EventSite8 oneSite = new EventSite8();  
        EventSite8 anotherSite = new EventSite8();  
        oneSite.setSiteNumber(101);  
        anotherSite.setSiteNumber(202);  
        System.out.print("The number of one site is ");  
        System.out.println(oneSite.getSiteNumber());  
        System.out.print("Headquarters located at ");  
        System.out.println(oneSite.HEADQUARTERS);  
        System.out.print("The number of another site is ");  
        System.out.println(anotherSite.getSiteNumber());  
        System.out.print("Headquarters located at ");  
        System.out.println(anotherSite.HEADQUARTERS);  
    }  
}
```

**Figure 4-19** DemoClassVar program

7. Save the file as **DemoClassVar.java** in the Chapter.04 folder. Compile and test the program. Figure 4-20 shows the program's output.



```

A:\Chapter.04>javac DemoClassVar.java
A:\Chapter.04>java DemoClassVar
The number of one site is 191
Headquarters located at Crystal Lake, IL
The number of another site is 202
Headquarters located at Crystal Lake, IL
A:\Chapter.04>

```

Figure 4-20 Output of the DemoClassVar program

## USING AUTOMATICALLY IMPORTED, PREWRITTEN CONSTANTS AND METHODS

There are many times when you need to create classes from which you will instantiate objects. You can create an Employee class with fields appropriate for describing employees and their functions, and an Inventory class with fields appropriate for whatever type of item it is that you manufacture. There are, however, many classes that a wide variety of programmers need. Rather than having each Java programmer “reinvent the wheel,” the creators of Java created nearly 500 classes for you to use in your programs.

You already used several of the prewritten classes without being aware of it. System, Character, Boolean, Byte, Short, Integer, Long, Float, and Double are actually classes from which you can create objects. These classes are stored in a **package**, or a **library of classes**, which is simply a folder that provides a convenient grouping for classes. There are many Java packages containing classes that are available only if you explicitly name them within your program, but the group of classes that contains the previously listed classes is used so frequently that it is available automatically to every program you write. The package that is implicitly imported into every Java program is named **java.lang**. The classes it contains are the **fundamental classes**, or basic classes, as opposed to the **optional classes** that must be explicitly named.



You will begin to import optional classes explicitly later in this chapter.

Tip

The class `java.lang.Math` contains constants and methods that you can use to perform common mathematical functions. All of the constants and methods in the Math class are **static**—they are class variables and class methods. A commonly used constant is `PI`.

Within the `Math` class, the declaration for `PI` is `public final static double PI = 3.14159265358979323846;`. Notice that `PI` is:

- `public`, so any program can access it directly
- `final`, so it cannot be changed
- `static`, so only one copy exists
- `double`, so it holds a large floating-point value



In geometry, `PI` is an approximation of the value of the ratio of the circumference of a circle to its diameter.



Another useful constant is `E`, which represents the base of natural logarithms. Its definition is `public final static double E = 2.7182818284590452354;`.

You can use the value of `PI` within any program you write by referencing the full package path in which `PI` is defined; for example `areaOfCircle = java.lang.Math.PI * radius * radius;`. However, the `Math` class is imported automatically into your programs, so if you simply reference `Math.PI`, Java will recognize this code as a shortcut to the full package path. Therefore, the preferred (and simpler) statement is `areaOfCircle = Math.PI * radius * radius;`.

In addition to constants, there are many useful methods available within the `Math` class. For example, the `Math.max()` method returns the larger of two values, and the method `Math.abs()` returns the absolute value of a number. The statement `largerValue = Math.max(32, 75)` results in `largerValue` assuming the value 75, and the statement `posVal = Math.abs(-245);` results in `posVal` assuming the value 245. Table 4-1 lists some common `Math` class methods.

**Table 4-1** Common `Math` class methods

Method	Meaning
<code>abs(x)</code>	Absolute value of <code>x</code>
<code>acos(x)</code>	Arccosine of <code>x</code>
<code>asin(x)</code>	Arcsine of <code>x</code>
<code>atan(x)</code>	Arctangent of <code>x</code>
<code>atan2(x,y)</code>	Theta component of the polar coordinate ( <code>r,theta</code> ) that corresponds to the Cartesian coordinate <code>x,y</code>
<code>ceil(x)</code>	Smallest integral value not less than <code>x</code> (ceiling)

Table 4-1 Common Math class methods (continued)

Method	Meaning
cos(x)	Cosine of x
exp(x)	Exponent, where e is the base of the natural logarithms
floor(x)	Largest integral value not greater than x
log(x)	Natural logarithm of x
max(x,y)	Larger of x and y
min(x,y)	Smaller of x and y
pow(x,y)	x raised to the y power
random()	Random double number between 0.0 and 1.0
rint(x)	Closest integer to x (x is a double, and the return value is expressed as a double)
round(x)	Closest integer to x (where x is a float or double, and the return value is an integer or long)
sin(x)	Sine of x
sqrt(x)	Square root of x
tan(x)	Tangent of x



Because all constants and methods in the Math class are classwide, there is no need to create an instance. You cannot instantiate objects of type Math because the constructor for the Math class is private and your programs cannot access the constructor.

Unless you are a mathematician, you won't use many of these Math class methods, and it is unwise to do so unless you understand their purposes. For example, because it is illegal to take the square root of a negative number, the method call `imaginaryNumber = Math.sqrt(-12);` causes a compiler error and does not execute.

Next you will use the Math class to perform some basic calculations.

#### To write a program that uses some Math class methods:

1. Open a new file in your text editor. Type the DemoMath class header `public class DemoMath`. On a new line, type the opening curly brace for the class, and then press **[Enter]**.
2. Type the `main()` method header `public static void main(String[] args)`, press **[Enter]**, type the opening curly brace for the `main()` method, and then press **[Enter]** to create a new line.
3. Create a double variable named `val` by typing `double val = 26.9;`, and then press **[Enter]**.



4. Type the following statement that displays the value on the screen:  
`System.out.println("The value is " + val);`
5. On separate lines, type the following statements to demonstrate the Math class methods:

```
System.out.print("Absolute value of val is ");  
System.out.println(Math.abs(val));  
System.out.print("Absolute value of -val is ");  
System.out.println(Math.abs(-val));  
System.out.print("The square root of val is ");  
System.out.println(Math.sqrt(val));  
System.out.print("Val rounded is ");  
System.out.println(Math.round(val));  
System.out.print("A random number is ");  
System.out.println(Math.random());  
System.out.print("8.0 raised to the 2 power is ");  
System.out.println(Math.pow(8.0, 2));
```



The expression `-val` means “negative val.” The minus sign (`-`) used in this manner is a unary or single-argument operator. You will learn more about unary operators in Chapter 5.

6. Add closing curly braces for the `main()` method and for the class.
7. Save the program as **DemoMath.java** in the Chapter.04 folder on your Student Disk, compile the program, run it, and then compare your results to Figure 4-21.

```
Command Prompt  
A:\Chapter_04>java DemoMath  
the value is 26.9  
Absolute value of val is 26.9  
Absolute value of -val is 26.9  
The square root of val is 5.186520991955976  
Val rounded is 27  
A random number is 0.44641977156695933  
8.0 raised to the 2 power is 64.0  
A:\Chapter_04>
```

**Figure 4-21** Output of the DemoMath program

8. Add additional statements that demonstrate any of the other Math methods that you might use in your programs. Save, compile, and test the program again.

## USING PREWRITTEN IMPORTED METHODS

Java contains hundreds of classes, only a few of which—such as `java.lang`—are included automatically in the programs you write. To use any of the other prewritten classes, you must use one of three methods:

- Use the entire path with the class name.
- Import the class.
- Import the package which contains the class you are using.

For example, the `java.util` class package contains useful methods that deal with dates and times. Within this package, a class named `Date` is defined. You can instantiate an object of type `Date` from this class by using the full class path, as in `java.util.Date myAnniversary = new java.util.Date();`. Alternately, you can shorten the declaration of `myAnniversary` to `Date myAnniversary = new Date();` by including `import java.util.Date;` as the first line in your program. An import statement allows you to abbreviate lengthy class names by notifying the Java program that when you use `Date`, you mean the `java.util.Date` class. You must place any import statement you use before any executing statement in your program. That is, you can have a blank line or a comment line—but nothing else—prior to an import statement.



`Date` is not a reserved word; it is a class you are importing. If you do not want to import the Java utility's `Date` class, you are free to write your own `Date` class.

An alternative to importing a class is to import an entire package of classes. You can use the asterisk (\*) as a **wildcard symbol** to represent all the classes in a package. Therefore, the import statement `import java.util.*;` imports the `Date` class and any other `java.util` classes as well. There is no disadvantage to importing the extra classes, and you will most commonly see the wildcard method in professionally written Java programs.



You cannot use the Java language wildcard exactly like a DOS or UNIX wildcard because you cannot import all the Java classes with `import java.*;`. The Java wildcard works only with specific packages such as `import java.util.*;` or `import java.lang.*;`.



Notice that the import statement ends with a semicolon. The import statement does not move the entire imported class or package into your program as its name implies. Rather, it simply notifies the program that you will be using the data and method names that are part of the imported class or package.

The `Date` class has several constructors. For example, if you construct a `Date` object with five integer arguments, they become the year, month, day, hour, and minutes. A `Date` object constructed with three integer arguments assumes the arguments to be the year, month, and day,

and the time is set to midnight. The constructor that takes no argument assigns the current moment to a `Date` object. The current moment is the number of milliseconds that have elapsed since midnight, January 1, 1970. Therefore, the statement `Date myAnniversary = new Date()` assigns a value that is a very large 12- or 13-digit number to the `myAnniversary` variable. You can retrieve this number with a method named `getTime()`. The statement `System.out.println("Milliseconds since 1/1/70 are " + myAnniversary.getTime());` results in the output `Milliseconds since 1/1/70 are 1066233611927` when the program is run at midnight on October 15, 2003.



If you set the hours in a `Date` object, a 24-hour clock is assumed—for example, 13 is 1 P.M.

Although it is interesting, the number of milliseconds elapsed since 1970 is not a useful piece of information for most people. Fortunately, the `Date` class does contain a variety of methods such as `setMonth()`, `getMonth()`, `setDay()`, `getDay()`, `setYear()`, and `getYear()`, which supply more-useful information. The program shown in Figure 4-22 shows the values of two dates being set and retrieved.

```
import java.util.*;
public class DemoDate
{
    public static void main(String[] args)
    {
        Date today = new Date();
        Date birthDay = new Date(82,6,14);
        System.out.println(today);
        System.out.print("Current month is ");
        System.out.println(today.getMonth());
        System.out.print("Current day is ");
        System.out.println(today.getDate());
        System.out.print("Current year is ");
        System.out.println(today.getYear());
        System.out.print("Birth month is ");
        System.out.println( birthDay.getMonth());
        System.out.print("Birth day is ");
        System.out.println(birthDay.getDate());
        System.out.print("Birth year is ");
        System.out.println(birthDay.getYear());
    }
}
```

**Figure 4-22** DemoDate program

You can perform arithmetic using dates. For example, if `toDay` is declared to hold today's date with `Date toDay = new Date();`, then you can use the following code to find out the due date of a bank certificate that matures in 180 days by adding 180 to the day part of the `Date` object:

```
toDay.setDate(toDay.getDate() + 180);
System.out.println("In 180 days it will be " + toDay);
```



The compiler will interpret an incorrect date, such as March 32, as being April 1. This makes many calculations with dates easier. For example, if you bill a customer on August 30 and allow 10 days for payment, you can add 10 to the billing day, and the compiler will understand August 40 to be September 9.



For information about time, including how leap years and leap seconds are calculated, go to the U.S. Naval Observatory Web site at <http://tycho.usno.navy.mil>.

Any year that you use with these `Date` class methods is a value that is 1900 less than the actual year. For example, 82 means 1982 and 105 means 2005. The month is a value from 0 through 11; January is 0, February is 1, and so on. You must be aware of this value organization when analyzing the meaning of a date.

Next you will use the `Date` class by declaring some `Date` variables and keeping track of the length of time it takes for the program to run.

#### To write a program that uses the `Date` class:

1. Open a new file in your text editor.
2. For the first line in the file, type `import java.util.*;`, press **[Enter]**, and then indent the line two spaces.
3. Begin a `DemoDate2` class with the header `public class DemoDate2`. Press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.
4. On the new line, indent two more spaces, and then type the following `main()` class header: `public static void main(String[] args)`. On a new line, enter the opening curly brace for the `main()` method, and then press **[Enter]**.
5. Declare a variable named `startTime`, and then assign it the current time by typing `Date startTime = new Date();`.
6. Declare another variable to hold the day your Java programming class began, for example, `Date classStart = new Date(103,7,25);` (where 103,7,25 in this example is August 25, 2003). Don't forget that the current year is 1900 less than the actual year and that the months are numbered 0 through 11.

7. Display the current date and the class start date by typing the following:

```
System.out.println("The current date is " + startTime);
System.out.println("The class started on " + classStart);
```

8. Save the file as **DemoDate2.java** in the Chapter.04 folder on your Student Disk.

Now enter a statement to print the time it takes to run this program. You will create a new `endTime` object that will hold the current date and time of its creation. Depending on the speed of the computer processor you are using, this time should be a few hundred milliseconds later than it was when the program started. The calculation involves using the `getTime()` method for the `endTime` and `startTime` objects and displaying the difference between the two values.

#### To use the `getTime()` method:

1. Open **DemoDate2.java**, if necessary, and then change the class name to **DemoDate3.java**.
2. Position the insertion point after the last print statement, press **[Enter]**, and then type the following code to include the `getTime()` method:

```
Date endTime = new Date();
System.out.print("Time elapsed is ");
System.out.print(endTime.getTime() -
    startTime.getTime());
System.out.println(" milliseconds");
```

3. Add the closing curly brace for the `main()` method as well as the closing curly brace for the program.
4. Save the program as **DemoDate3.java**, and then compile and test the program.



When you compile the `DemoDate3.java` program, you might receive the following error from the compiler: `DemoDate3.java uses or overrides a deprecated API. Recompile with "-deprecation" for details. 1 warning.` This warning indicates that your program compiled successfully. A deprecated API simply indicates that your program uses something that has been improved in subsequent versions of Java. If you want to see information about the methods that are deprecated, then you can recompile the `DemoDate3` program using `javac -deprecation DemoDate3.java`.

5. Add some extra `println()` statements to the program, and save, compile, and run the program again. Does the program take longer to execute?

## LEARNING ABOUT GREGORIAN CALENDARS

The preceding Help text indicates that `Date` is a deprecated API. Most of the methods in the `Date` class have been replaced by the `GregorianCalendar` class to support the Gregorian calendar, the calendar used in most of the western world. There are seven constructors for `GregorianCalendar` objects. The default creates a calendar with the current date and time in the default locale. You can use other constructors to specify the year, month, day, hour, minute, and second. You create a calendar object with the default constructor `GregorianCalendar calendar = new GregorianCalendar();`. To calculate time in milliseconds, you can use the `getTimeInMillis()` method, as in `calendar.getTimeInMillis()`.

Information such as the day, month, and year can be retrieved from a `GregorianCalendar` object by using a class `get()` method, and then specifying what you want as an argument. All values returned are of type `int`. For example, you could get the day of the year with the statement `int dayOfYear = calendar.get(calendar.DAY_OF_YEAR);`. Some of the possible arguments to the `get()` method are shown in Table 4-2.

**Table 4-2** Some possible returns from the `GregorianCalendar` `get()` method

Arguments	Values returned by <code>get()</code>
<code>DAY_OF_YEAR</code>	A value of 1 to 366
<code>DAY_OF_MONTH</code>	A value from 1 to 31
<code>DAY_OF_WEEK</code>	<code>SUNDAY</code> , <code>MONDAY</code> , ..., <code>SATURDAY</code> , corresponding to values of 1 to 7
<code>YEAR</code>	The current year, for example, 2003
<code>MONTH</code>	<code>JANUARY</code> , <code>FEBRUARY</code> , ..., <code>DECEMBER</code> , corresponding to values of 0 to 11
<code>HOURL</code>	A value of 1 to 12 being the current hour in the A.M. or P.M.
<code>AM_PM</code>	A.M. or P.M., which correspond to values of 0 to 1
<code>HOURL_OF_DAY</code>	A value of 0 to 23
<code>MINUTE</code>	The current minute in the current hour, a value of 0 to 59
<code>SECOND</code>	The second in the current minute, a value of 0 to 59
<code>MILLISECOND</code>	The millisecond in the current second, a value of 0 to 999

Next you will construct a program using the `GregorianCalendar` class and using some of the arguments to the `GregorianCalendar` `get()` method.

### To write a program that uses the `GregorianCalendar` class:

1. Open a new file in your text editor.
2. For the first line in the file, type `import java.util.*;`, press **[Enter]**, and then indent the line two spaces.
3. Begin by typing a header class `public class Birthdate`. Press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again.

4. On the new line, indent two more spaces, and then type the following main class header: **public static void main(String[] args)**. Type the opening curly brace for the main() method, and then press **[Enter]**.
5. Declare integer values to hold a birthdate year, month, and day. Place each on a separate line by pressing **[Enter]** after each integer declaration:

```
int ayear = 1940;  
int amonth = 0; //month is 0 to 11  
int aday = 31;
```

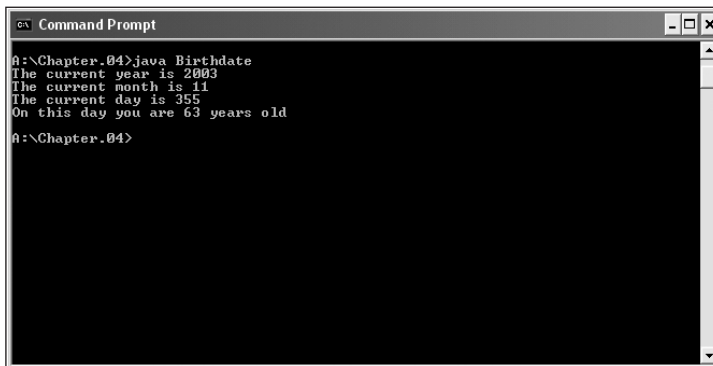
6. Create a new GregorianCalendar object acalendar as **GregorianCalendar acalendar = new GregorianCalendar(ayear,amonth,aday);**.
7. Press **[Enter]** and create a second new GregorianCalendar bcalendar as **GregorianCalendar bcalendar = new GregorianCalendar();**.
8. Press **[Enter]**, and then create three separate calls to the get() method, storing each value returned in a separate integer variable byear, bmonth, and bday:

```
int byear = bcalendar.get(bcalendar.YEAR);  
int bmonth = bcalendar.get(bcalendar.MONTH);  
int bday = bcalendar.get(bcalendar.DAY_OF_WEEK);
```

9. Display the current year, month, and day by typing the following:

```
System.out.println("The current year is " + byear);  
System.out.println("The current month is " + bmonth);  
System.out.println("The current day is " + bday);  
System.out.println("On this day you are " + (byear -  
    ayear) + " years old");
```

10. Finally, put the closing curly braces on separate lines to end the main() method and the Birthday class.
11. Save the file as **Birthday.java** in the Chapter.04 folder on your Student Disk. Compile and run the program. The output is shown in Figure 4-23. Note that your results may differ from the output shown in Figure 4-23.



```
Command Prompt  
A:\Chapter.04>java Birthday  
The current year is 2003  
The current month is 11  
The current day is 355  
On this day you are 63 years old  
A:\Chapter.04>
```

Figure 4-23 Output of the Birthday program

---

## CHAPTER SUMMARY

- A variable's scope is the portion of a program within which you can reference that variable. A variable comes into scope (comes into existence) when you declare it, and goes out of scope (ceases to exist) at the end of the block in which it is declared.
- A block is the code between a pair of curly braces. You can nest blocks within other blocks. Within a method, you can declare a variable with the same name multiple times as long as each declaration is in its own, nonoverlapping block. Within nested blocks, you cannot declare the same variable name more than once. If you declare a variable within a class and use the same variable name within a method of the class, then the variable used inside the method takes precedence, or overrides, the first variable.
- Overloading involves writing multiple methods with the same name but different argument lists. Methods that have identical argument lists but different return types are not overloaded; they are illegal.
- Constructor methods can receive arguments and be overloaded. If you explicitly create a constructor for a class, the automatically created constructor no longer exists.
- You store just one copy of a method for use with each object. You store separate copies of data fields for each object. The compiler accesses the correct object's data fields because you implicitly pass a `this` reference to class methods. Static methods do not have a `this` reference because they have no object associated with them. Static methods are also called class methods.
- Static class variables are those variables that are shared by every instantiation of a class.
- After a program is compiled, literal constants never change. Also, the values stored in symbolic constants never change. You create a symbolic constant by inserting the keyword `final` before a variable name. By convention, constant fields are written using all uppercase letters. A constant must be initialized with a value.
- Java contains nearly 500 prewritten classes which are stored in a package, which is simply a folder that provides a convenient grouping for classes. The package that is implicitly imported into every Java program is named `java.lang`. The classes it contains are the fundamental classes, as opposed to the optional classes, which must be explicitly named.
- The class `java.lang.Math` contains constants and methods that can be used to perform common mathematical functions. All of the constants and methods in the `Math` class are static—they are class variables and class methods. Common useful `Math` class methods include those used for finding an absolute value, taking a square root, and rounding. To use a prewritten class other than `java.lang`, you must use the entire path with the class name, import the class, or import the package that contains the class.



- An import statement allows you to abbreviate lengthy class names by notifying the Java program that when you use class names you are referring to those within the imported class. Any import statement you use must be placed before any executing statement in your program. An alternative to importing a class is to import an entire package of classes. To do so, you can use the asterisk (\*) as a wildcard symbol to represent all the classes in a package.
- The Date class has several constructors: one that takes no argument and assigns the current moment to a Date object; and others that take the date, or the date and time. The current moment is the number of milliseconds that have elapsed since midnight, January 1, 1970. You can retrieve this number with the getTime() method. The Date class contains a variety of other methods, such as setMonth(), getMonth(), setDay(), getDay(), setYear(), and getYear(), which supply more-useful information.
- The GregorianCalendar class is the calendar generally used in the western world. The GregorianCalendar class has seven constructors and a number of get() methods to define and manipulate dates and time.

---

## REVIEW QUESTIONS

1. The code between a pair of curly braces in a method is a \_\_\_\_\_.
  - a. function
  - b. block
  - c. brick
  - d. sector
2. When a block exists within another block, the blocks are \_\_\_\_\_.
  - a. structured
  - b. nested
  - c. sheltered
  - d. illegal
3. The portion of a program within which you can reference a variable is the variable's \_\_\_\_\_.
  - a. range
  - b. space
  - c. domain
  - d. scope

4. You can declare a variable with the same name multiple times \_\_\_\_\_.
  - a. within a statement
  - b. within a block
  - c. within a method
  - d. never
5. If you declare a variable within a class, and declare and use the same variable name within a method of the class, \_\_\_\_\_.
  - a. the variable used inside the method takes precedence
  - b. the class variable takes precedence
  - c. they become the same variable with the same memory address
  - d. an error will occur
6. A method variable will \_\_\_\_\_ a class variable with the same name.
  - a. acquiesce to
  - b. destroy
  - c. override
  - d. alter
7. Overloaded methods must have the same \_\_\_\_\_.
  - a. name
  - b. number of arguments
  - c. argument names
  - d. type of argument
8. If a method is written to receive a double argument, and you pass an integer to the method, then the method will \_\_\_\_\_.
  - a. work correctly; the integer will be promoted to a double
  - b. work correctly; the integer will remain an integer
  - c. execute; but any output will be incorrect
  - d. not work; an error message will be issued
9. A constructor \_\_\_\_\_ arguments.
  - a. can receive
  - b. cannot receive
  - c. must receive
  - d. can receive a maximum of 10
10. A constructor \_\_\_\_\_ overloaded.
  - a. can be
  - b. cannot be

- c. must be
  - d. is always automatically
11. Usually you want each instantiation of a class to have its own copy of \_\_\_\_\_.
- a. the data fields
  - b. the class methods
  - c. both of the above
  - d. none of the above
12. If you create a class and instantiate two objects, you usually store \_\_\_\_\_ for use with the objects.
- a. one copy of each method
  - b. two copies of the same method
  - c. two different methods
  - d. data only, not methods
13. The **this** reference \_\_\_\_\_.
- a. can be used implicitly
  - b. must be used implicitly
  - c. must not be used implicitly
  - d. must not be used
14. Methods that you associate with individual objects are \_\_\_\_\_.
- a. **private**
  - b. **public**
  - c. **static**
  - d. **nonstatic**
15. Variables that are shared by every instantiation of a class are \_\_\_\_\_.
- a. class variables
  - b. **private** variables
  - c. **public** variables
  - d. illegal
16. The keyword **final** in a variable declaration indicates \_\_\_\_\_.
- a. the end of the program
  - b. a static field
  - c. a symbolic constant
  - d. that no more variables will be declared in the program

17. Java classes are stored in a folder or \_\_\_\_\_.  
a. packet  
b. package  
c. bundle  
d. gaggle
18. Which of the following statements determines the square root of a number and assigns it to the variable `s`?  
a. `s = sqrt(number);`  
b. `s = Math.sqrt(number);`  
c. `number = sqrt(s);`  
d. `number = Math.sqrt(s);`
19. A `GregorianCalendar` object can be created with one of \_\_\_\_\_ constructors.  
a. two  
b. four  
c. seven  
d. nine
20. The `get()` method using the `DAY_OF_WEEK` argument returns \_\_\_\_\_.  
a. SUNDAY to SATURDAY  
b. a value from 0 to 6  
c. a value from 1 to 7  
d. a value of 1 to 6

---

## EXERCISES

1. a. Create a class named `Commission` that includes three variables: a double sales figure, a double commission rate, and an integer commission rate. Create two overloaded methods named `computeCommission()`. The first method takes two double arguments representing sales and rate, multiplies them, and then displays the results. The second method takes two arguments: a double sales figure and an integer commission rate. This method must divide the commission rate figure by 100.0 before multiplying by the sales figure and displaying the commission. Supply appropriate values for the variables, and write a `main()` method that tests each overloaded method. Save the program as **Commission.java** in

the Chapter.04 folder on your Student Disk, and then compile and test the program.

- b. Add a third overloaded method to the Commission program you created in Exercise 1a. The third overloaded method takes a single argument representing sales. When this method is called, the commission rate is assumed to be 7.5 percent and the results are displayed. To test this method, add an appropriate call in the Commission program's `main()` method. Save the program as **Commission2.java** in the Chapter.04 folder on your Student Disk, and then compile and test it.
2. Create a class named `Pay` that includes five double variables that hold hours worked, rate of pay per hour, withholding rate, gross pay, and net pay. Create three overloaded `computeNetPay()` methods. Gross pay is computed as hours worked, multiplied by pay per hour. When `computeNetPay()` receives values for hours, pay rate, and withholding rate, it computes the gross pay and reduces it by the appropriate withholding amount to produce the net pay. When `computeNetPay()` receives two arguments, the withholding rate is assumed to be 15 percent. When `computeNetPay()` receives one argument, the withholding rate is assumed to be 15 percent, and the hourly rate is assumed to be 4.65. Write a `main()` method that tests all three overloaded methods. Save the program as **Pay.java** in the Chapter.04 folder on your Student Disk.
3.
  - a. Create a class named `Household` that includes data fields for the number of occupants and the annual income, as well as methods named `setOccupants()`, `setIncome()`, `getOccupants()`, and `getIncome()` that set and return those values, respectively. Additionally, create a constructor that requires no arguments and automatically sets the occupants field to 1 and the income field to 0. Save this file as **Household.java** in the Chapter.04 folder on your Student Disk. Create a program named `TestHousehold` that demonstrates that each method works correctly. Save the file as **TestHousehold.java** in the Chapter.04 folder on your Student Disk.
  - b. Create an additional overloaded constructor for the `Household` class you created in Exercise 3a. This constructor receives an integer argument and assigns the value to the occupants field. Add any needed statements to `TestHousehold` to ensure that the overloaded constructor works correctly, save it, and then test it.
  - c. Create a third overloaded constructor for the `Household` class you created in Exercises 3a and 3b. This constructor receives two arguments, the values of which are assigned to the occupants and income fields, respectively. Alter the `TestHousehold` program to demonstrate that each version of the constructor works properly. Save the program, and then compile and test it.
4. Create a class named `Box` that includes integer data fields for length, width, and height. Create three constructors that require one, two, and three arguments, respectively. When one argument is used, assign it to length, assign zeros to height and width, and print "Line created". When two arguments are used, assign them to length and width, assign zero to height, and print "Rectangle created". When three arguments are used, assign them to the three variables and print "Box created".

Save this file as **Box.java** in the Chapter.04 folder of your Student Disk. Create a program named **TestBox** that demonstrates that each method works correctly. Save the test file as **TestBox.java** in the Chapter.04 folder on your Student Disk.

5. What is the result when you compile and run the following code? Why?

```
class Scope
{
    int scopeInt = 1;
    void scopeDisplay()
    {
        int scopeInt = 10;
        System.out.println("scopeInt = " + scopeInt);
    }
    public static void main(String[] args)
    {
        Scope scopeExercise = new Scope();
        scopeExercise.scopeDisplay();
    }
}
```

6. a. What is the result when you compile and run the following code? Why?

```
class Overload
{
    public static void main(String[] args)
    {
        Overload overloadExercise = new Overload();
        overloadExercise.methodOv();
        overloadExercise.methodOv(6.1, 3);
    }
    void methodOv()
    {
        System.out.println("no arguments");
    }
    void methodOv(double dblArg, int intArg)
    {
        System.out.println("dblArg = " + dblArg + "intArg = " +
            intArg);
    }
}
```

- b. What happens when you compile and run the program shown in Exercise 6a if you replace the line `overloadExercise.methodOv(6.1, 3);` with `overloadExercise.methodOv(6, 3);`, and why?
- c. What happens if you change the program shown in Exercise 6a as follows, and why?

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

```

    {
        Overload overloadExercise = new Overload();
        overloadExercise.methodOv(6.1, 3.2);
    }
    void methodOv(double dblArg, float fltArg)
    {
        System.out.println("dblArg = " + dblArg + " fltArg = "
+ fltArg);
    }
    void methodOv(float fltArg, double dblArg)
    {
        System.out.println("dblArg = " + dblArg + " fltArg = "
+ fltArg);
    }
}

```

- d. If the program shown in Exercise 6c results in a compile error, how would you fix the program so it compiles and runs successfully?
7. Create a class named **Shirt** with data fields for collar size and sleeve length. Include a constructor method that takes arguments for each field. Also include a String class variable named **material** and initialize it to "cotton". Write a program named **TestShirt** to instantiate three **Shirt** objects with different collar sizes and sleeve lengths, and then display all the data, including material, for each shirt. Save both the **Shirt.java** and **TestShirt.java** programs in the Chapter.04 folder of your Student Disk.
8. Create a class named **CheckingAccount** with data fields for an account number and a balance. Include a constructor method that takes arguments for each field. Include a double class variable that holds a value for the minimum balance required before a monthly fee is applied to the account. Set the minimum balance to 200.00. Write a program named **TestAccount** in which you instantiate two **CheckingAccount** objects and display the account number, balance, and minimum balance without fee for both accounts. Save both the **CheckingAccount.java** and **TestAccount.java** programs in the Chapter.04 folder on your Student Disk.
9. Write a Java program to determine the answers for each of the following:
  - a. the square root of 30
  - b. the sine and cosine of 100
  - c. the value of the floor, ceiling, and round of 44.7
  - d. the larger and the smaller of the character K and the integer 70
 Save the file as **MathTest.java** in the Chapter.04 folder on your Student Disk.
10. Write a program to calculate how many milliseconds it is from today until the first day of next summer (assume that this date is June 21). Use the **Date** class. Save the file as **Summer.java** in the Chapter.04 folder on your Student Disk.

11. Write a program to calculate how many days it is from today until the end of the current year. Use the `Date` class. Save the file as **YearEnd.java** in the Chapter.04 folder on your Student Disk.

12. What is the result when you compile and run the following code, and why?

```
public class MathEx6
{
    public static void main(String[] args)
    {
        System.out.println(Math.round(1.49));
        System.out.println(Math.round(1.50));
        System.out.println(Math.round(-1.49));
        System.out.println(Math.round(-1.50));
    }
}
```

13. What is the result when you compile and run the following code, and why?

```
public class MathEx13
{
    public static void main(String[] args)
    {
        System.out.println(Math.ceil(1.49));
        System.out.println(Math.ceil(1.50));
        System.out.println(Math.ceil(-1.49));
        System.out.println(Math.ceil(-1.50));
    }
}
```

14. What is the result when you compile and run the following code, and why?

```
public class MathEx14
{
    public static void main(String[] args)
    {
        System.out.println(Math.floor(1.49));
        System.out.println(Math.floor(1.50));
        System.out.println(Math.floor(-1.49));
        System.out.println(Math.floor(-1.50));
    }
}
```

15. Modify the `Employee` class shown in Figure 4-17 by changing the class name to `EmployeeWithDate`. Then change the `showCompanyID()` method so it shows the current date, in addition to the employee number and company ID. Save the file as **EmployeeWithDate.java** in the Chapter.04 folder on your Student Disk. Then write a program that creates and displays two or more `EmployeeWithDate` objects. Save this new program as **UseEmployeeWithDate.java** in the Chapter.04 folder on your Student Disk.



16. Write a program to calculate how many milliseconds it is from today until the first day of next summer (assume that this date is June 21). Use the `GregorianCalendar` class. Save the file as **Summer2.java** in the Chapter.04 folder on your Student Disk.
17. Write a program to calculate how many days it is from today until the end of the current year. Use the `GregorianCalendar` class. Save the file as **YearEnd2.java** in the Chapter.04 folder on your Student Disk.
18. Each of the following files in the Chapter.04 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, save `DebugThree1.java` as `FixDebugFour1.java`.
  - a. `DebugFour1.java`
  - b. `DebugFour2.java`
  - c. `DebugFour3.java`
  - d. `DebugFour4.java`

## CASE PROJECT



The Pool Associates operates a business that offers a variety of services to the general public who own swimming pools. Each year, Pool Associates cleans local pools and fills those pools when needed. Because swimming pools require a different amount of time to service, your job is to write a program that calculates the amount of time it will take to fill the pools with water. Then a reasonable estimate of fill-up time can be made before the job begins. This will enable Pool Associates to charge for pool fill-up on the basis of estimated hours. The table below gives some necessary parameters for estimating the fill-up time for a small pool and a large pool. Calculate the fill-up time for a small pool and a large pool. A small pool is considered 20 by 12 by 4 feet, and a large pool 30 by 20 by 10 feet. Save the program as **Swimming.java** in the Chapter.04 folder on your Student Disk.

Problem Parameters
Pool Volume $L * W * D$
Pool Capacity $L * W * D * CAPACITY$
Time to Fill $L * W * D * CAPACITY / (RATE\_OF\_FLOW * 60)$
RATE_OF_FLOW 50.0 gal/min
CAPACITY 7.5 gal/cubic foot
L – pool length
W – pool width
D – pool depth

