

More Object Concepts

In this chapter, you will:

- ⦿ Understand blocks and scope
- ⦿ Overload a method
- ⦿ Avoid ambiguity
- ⦿ Create and call constructors with parameters
- ⦿ Use the `this` reference
- ⦿ Use static fields
- ⦿ Use automatically imported, prewritten constants and methods
- ⦿ Use composition and nest 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 method shown in Figure 4-1 contains two blocks. The first block contains another, so it is an example of an **outside block** (also called an **outer block**). It begins immediately after the method declaration and ends at the end of the method. The second block is called the **inside block** or **inner block**. It is contained within the second set of curly braces and contains two executable statements: the declaration of `anotherNumber` and a `println()` statement. The inside block is **nested**, or contained entirely within, the outside block.

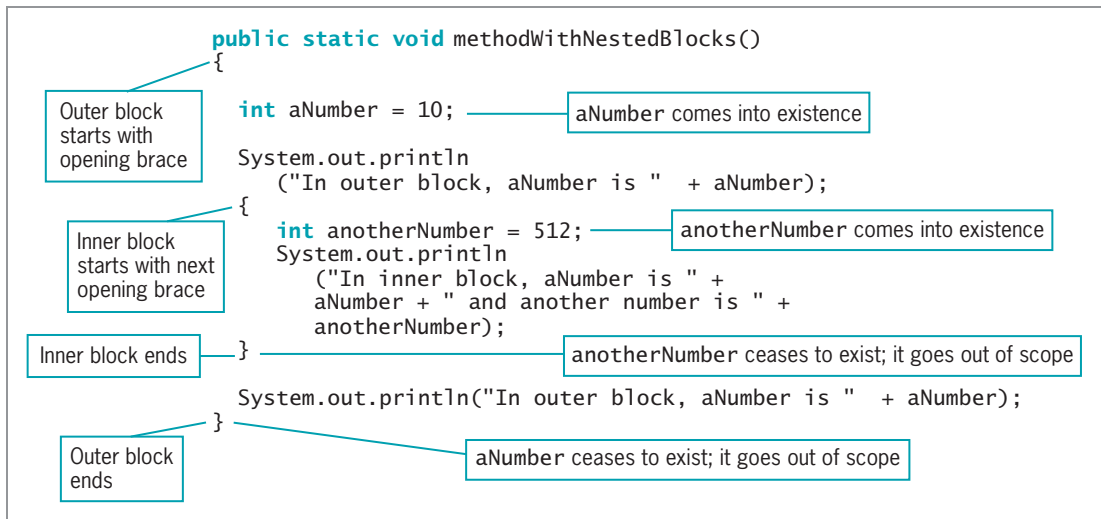


Figure 4-1 A method with nested blocks

A block can exist entirely within another block or entirely outside and separate from another block, but blocks can never overlap. For example, if a method contains two opening curly braces, indicating the start of two blocks, the next closing curly brace always closes the inner (second) block—it cannot close the outer block because that would make the blocks overlap. Another way to state this concept is that whenever you encounter a closing brace that ends a block, it always closes the most recently opened block.

When you declare a variable, you cannot refer to that variable outside its block. As you learned in Chapter 3, the portion of a program within which you can refer to 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 `methodWithNestedBlocks()` 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 in the inner block and can be used anywhere in the method. The variable `anotherNumber` comes into existence within the inner block; `anotherNumber` goes out of scope when the inner block ends and cannot be used beyond its block. Figure 4-2 shows the output when the method in Figure 4-1 is called from another method.



```

C:\Java>java TestMethodWithNestedBlocks
In outer block, aNumber is 10
In inner block, aNumber is 10 and another number is 512
In outer block, aNumber is 10
C:\Java>

```

Figure 4-2 Output produced by application that uses `methodWithNestedBlocks()`



The program that produces the output shown in Figure 4-2 is stored in the `CodelnFigures` folder in your downloadable student files.

You cannot use a data item that is not in scope. For example, Figure 4-3 shows a method that contains two blocks and some shaded, invalid statements. The opening and closing braces for each block are vertically aligned. 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.

```

public static void methodWithInvalidStatements()
{
    aNumber = 75; // Illegal statement; this variable has not been declared yet
    int aNumber = 22;
    aNumber = 6;
    anotherNumber = 489; // Illegal statement; this variable has not been declared yet
    {
        anotherNumber = 165; // Illegal statement; this variable still has not been declared
        int anotherNumber = 99;
        anotherNumber = 2;
    }
    aNumber = 50; // Illegal statement; this variable was declared in the inner block
    anotherNumber = 34; // and has gone out of scope here
}
aNumber = 29; // Illegal statement; this variable has gone out of scope

```

Figure 4-3 The `methodWithInvalidStatements()` method

The first assignment statement in the first, outer block in Figure 4-3, `aNumber = 75;`, is invalid because `aNumber` has not been declared yet. Similarly, the statements that attempt to assign 489 and 165 to `anotherNumber` are invalid because `anotherNumber` has not been declared yet. After `anotherNumber` is declared, it can be used for the remainder of the inner block, but the statement that attempts to assign 34 to it is outside the block in which `anotherNumber` was declared. The last shaded statement in Figure 4-3, `aNumber = 29;`, does not work because it falls outside the block in which `aNumber` was declared; it actually falls outside the entire `methodWithInvalidStatements()` method.

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 4-4 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 void twoDeclarations()
{
    {
        int someVar = 7;
        System.out.println(someVar);
    }
    {
        int someVar = 845;
        System.out.println(someVar);
    }
}

```

Don't declare blocks for no reason. A new block starts here only to demonstrate scope.

This variable will go out of scope at the next closing curly brace.

This variable is totally different from the one in the previous block even though their identifiers are the same.

Figure 4-4 The `twoDeclarations()` method

You cannot declare the same variable name more than once within a block, even if a block contains other blocks. When you declare a variable more than once in a block, you are attempting to **redeclare the variable**—an illegal action. For example, in Figure 4-5, 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 void invalidRedeclarationMethod()
{
    int aValue = 35;
    int aValue = 44;
    {
        int anotherValue = 0;
        int aValue = 10;
    }
}

```

Invalid redeclaration of `aValue` because it is in same block as the first declaration

Invalid redeclaration of `aValue`; even though this is a new block, this block is inside the first block

Figure 4-5 The `invalidRedeclarationMethod()`

Although you cannot declare a variable twice within the same block, you can declare a variable within one method of a class and use the same variable name within another method of the class. In this case, the variable declared inside each method resides in its own location in computer memory. When you use the variable's name within the method in which it is declared, it takes precedence over, or **overrides**, any other variable with the same name in another method. In other words, a locally declared variable always masks or hides another variable with the same name elsewhere in the class.

For example, consider the class in Figure 4-6. In the `main()` method of the `OverridingVariable` class, `aNumber` is declared and assigned the value 10. When the program calls `firstMethod()`, a new variable is declared with the same name but with a different memory address and a new value. The new variable exists only within `firstMethod()`, where it is displayed holding the value 77. After `firstMethod()` executes and the logic returns to the `main()` method, the original `aNumber` is displayed, containing 10. When `aNumber` is passed to `secondMethod()`, a copy is made within the method. This copy has the same identifier as the original `aNumber`, but a different memory address. So, within `secondMethod()`, when the value is changed to 862 and displayed, it has no effect on the original variable in `main()`. When the logic returns to `main()` after `secondMethod()`, the original value is displayed again. Examine the output in Figure 4-7 to understand the sequence of events.

```

public class OverridingVariable
{
    public static void main(String[] args)
    {
        int aNumber = 10;
        System.out.println("In main(), aNumber is " + aNumber);
        firstMethod();
        System.out.println("Back in main(), aNumber is " + aNumber);
        secondMethod(aNumber);
        System.out.println("Back in main() again, aNumber is " + aNumber);
    }
    public static void firstMethod()
    {
        int aNumber = 77;
        System.out.println("In firstMethod(), aNumber is "
            + aNumber);
    }
    public static void secondMethod(int aNumber)
    {
        System.out.println("In secondMethod(), at first " +
            "aNumber is " + aNumber);
        aNumber = 862;
        System.out.println("In secondMethod(), after an assignment " +
            "aNumber is " + aNumber);
    }
}

```

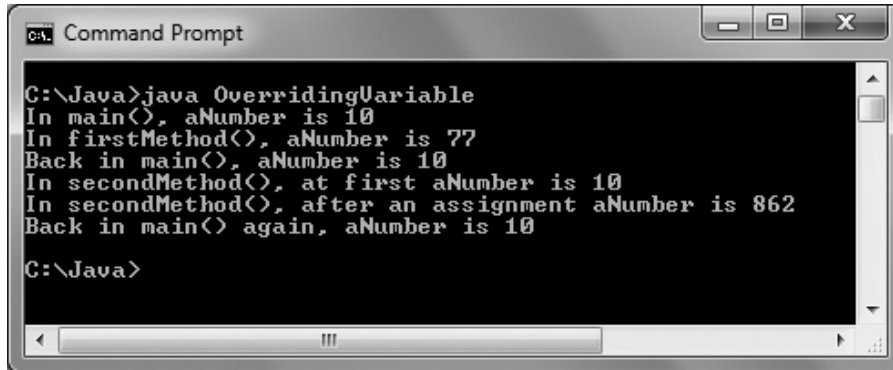
aNumber is declared in main().

Whenever aNumber is used in main(), it retains its value of 10.

This aNumber resides at a different memory address from the one in main(). It is declared locally in this method.

This aNumber also resides at a different memory address from the one in main(). It is declared locally in this method.

Figure 4-6 The `OverridingVariable` class



```
ca. Command Prompt
C:\Java>java OverridingVariable
In main(), aNumber is 10
In firstMethod(), aNumber is 77
Back in main(), aNumber is 10
In secondMethod(), at first aNumber is 10
In secondMethod(), after an assignment aNumber is 862
Back in main() again, aNumber is 10

C:\Java>
```

Figure 4-7 Output of the `OverridingVariable` application



Object-oriented programmers also use the term *override* when a child class contains a field or method that has the same name as one in the parent class. You will learn more about inheritance in the chapters *Introduction to Inheritance* and *Advanced Inheritance Concepts*.



You are familiar with local names overriding names defined elsewhere. If someone in your household is named Eric, and someone in the house next door is named Eric, members of your household who talk about Eric are referring to the local version. They would add a qualifier such as *Eric Johnson* or *Eric next door* to refer to the nonlocal version.

When they have the same name, variables within methods of a class override or hide the class's fields. Java calls this phenomenon **shadowing**; a variable that hides another shadows it. For example, Figure 4-8 shows an `Employee` class that contains two instance variables and three `void` methods. The `setValues()` method provides values for the two class instance fields. Whenever the method named `methodThatUsesInstanceAttributes()` is used with an `Employee` object, the instance values for `empNum` and `empPayRate` are used. However, when the other method, `methodThatUsesLocalVariables()`, is used with an `Employee` object, the local variable values within the method, 33333 and 555.55, shadow the class's instance variables. Figure 4-9 shows a short application that declares an `Employee` object and uses each method; Figure 4-10 shows the output.

```

public class Employee
{
    private int empNum;
    private double empPayRate;
    public void setValues()
    {
        empNum = 111;
        empPayRate = 22.22;
    }
    public void methodThatUsesInstanceAttributes()
    {
        System.out.println("Employee number is " + empNum);
        System.out.println("Pay rate is " + empPayRate);
    }
    public void methodThatUsesLocalVariables()
    {
        int empNum = 33333;
        double empPayRate = 555.55;
        System.out.println("Employee number is " + empNum);
        System.out.println("Pay rate is " + empPayRate);
    }
}

```

This method uses the class fields.

This method also uses the class fields.

This method uses the locally declared variables that happen to have the same names as the class fields.

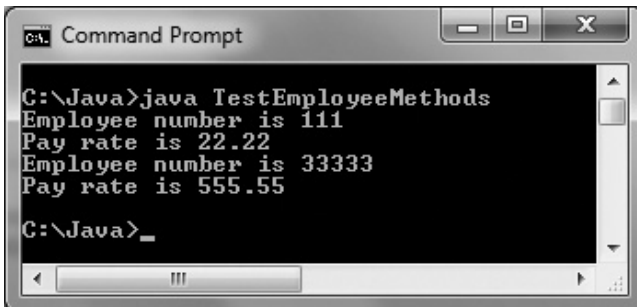
Figure 4-8 The Employee class

```

public class TestEmployeeMethods
{
    public static void main(String[] args)
    {
        Employee aWorker = new Employee();
        aWorker.setValues();
        aWorker.methodThatUsesInstanceAttributes();
        aWorker.methodThatUsesLocalVariables();
    }
}

```

Figure 4-9 The TestEmployeeMethods application



```

C:\Java>java TestEmployeeMethods
Employee number is 111
Pay rate is 22.22
Employee number is 33333
Pay rate is 555.55
C:\Java>_

```

Figure 4-10 Output of the TestEmployeeMethods application

In the `methodThatUsesLocalVariables()` method in Figure 4-8, the locally declared `empNum` and `empPayRate` are assigned 33333 and 55.55, respectively. These local variables are said to be **closer in scope** than the variables with the same name at the top of the class that are shadowed. When you write programs, you might choose to avoid confusing situations that arise when you give the same name to a class's instance field and to a local method variable. But, if you do use the same name, be aware that within the method, the method's local variable overrides the instance variable.



Programmers frequently use the same name for an instance field and a parameter to a method in the same class simply because it is the “best name” to use; in these cases, the programmer must use the `this` reference, which you will learn about later in this chapter.

It is important to understand the impact that blocks and methods have on your variables. Variables and fields with the same names represent different memory locations when they are declared within different scopes. After you understand the scope of variables, you can avoid many potential errors in your programs.

TWO TRUTHS & A LIE

Understanding Blocks and Scope

1. A variable ceases to exist, or goes out of scope, at the end of the block in which it is declared.
2. You cannot declare the same variable name more than once within a block, even if a block contains other blocks.
3. A class's instance variables override locally declared variables with the same names that are declared within the class's methods.

The false statement is #3. When they have the same name, variables within methods of a class override a class's instance variables.



You Do It

Demonstrating Scope

In this section, you create a method with several blocks to demonstrate block scope.

1. Start your text editor, and then open a new document, if necessary.
2. Type the first few lines for a class named `DemoBlock`:

(continues)

(continued)

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

3. Add a statement that displays the purpose of the program:

```
System.out.println("Demonstrating block scope");
```

4. On a new line, declare an integer named `x`, assign the value 1111 to it, and display its value:

```
int x = 1111;
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 named `y`, and display `x` and `y`. The value of `x` is 1111, and the value of `y` is 2222:

```
{
    int y = 2222;
    System.out.println("In second block x is " + x);
    System.out.println("In second block y is " + y);
}
```

6. On the next line, begin another new block. Within this new block, declare a new integer with the same name as the integer declared in the previous block; then display `x` and `y`. The value of `y` is 3333. Call a method named `demoMethod()`, and display `x` and `y` again. Even though you will include statements within `demoMethod()` that assign different values to `x` and `y`, the `x` and `y` displayed here are still 1111 and 3333:

```
{
    int y = 3333;
    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);
}
```

7. On a new line after the end of the block, type the following:

```
System.out.println("At the end x is " + x);
```

This last statement in the `main()` method displays the value of `x`, which is still 1111. Type a closing curly brace.

(continues)

(continued)

8. Finally, enter the following `demoMethod()` that creates its own `x` and `y` variables, 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);
}
```

9. Type the final closing curly brace, and then save the file as **DemoBlock.java**. At the command prompt, compile the file by typing the command **javac DemoBlock.java**. If necessary, correct any errors, and compile the program again.
10. Run the program by typing the command **java DemoBlock**. Your output should look like Figure 4-11. Make certain you understand how the values of `x` and `y` are determined in each line of output.
11. To gain a more complete understanding of blocks and scope, change the values of `x` and `y` in several locations throughout the program, and try to predict the exact output before resaving, recompiling, and rerunning the program.



```
C:\Java>java DemoBlock
Demonstrating block scope
In first block x is 1111
In second block x is 1111
In second block y is 2222
In third block x is 1111
In third block y is 3333
In demoMethod x is 8888
In demoMethod block y is 9999
After method x is 1111
After method block y is 3333
At the end x is 1111
C:\Java>
```

Figure 4-11 Output of the DemoBlock application

Overloading a Method

Overloading involves using one term to indicate diverse meanings. In Java, it more specifically means writing multiple methods in the same scope that have the same name but different parameter lists. The names used in the parameter lists do not matter; the lists must differ in parameter type, number of parameters, or both.

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, multiple methods share a name, and the compiler understands which one to use based on the arguments in the method call. For example, suppose you create a class method to apply a simple interest rate to a bank balance. The method is named `calculateInterest()`; it receives two `double` parameters—the balance and the interest rate—and displays the multiplied result. Figure 4-12 shows the method.

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

Figure 4-12 The `calculateInterest()` method with two `double` parameters

When an application calls the `calculateInterest()` method and passes two `double` values, as in `calculateInterest(1000.00, 0.04)`, the interest is calculated correctly as 4% of \$1000.00.

Assume, however, that different users want to calculate interest using different argument types. Some users who want to indicate an interest rate of 4% might use 0.04; others might use 4 and assume that it means 4%. When the `calculateInterest()` method is called with the arguments 1000.00 and 0.04, the interest is calculated correctly as 40.00. When the method is called using 1000.00 and 4, the method works because the integer argument is promoted to a `double`, but the interest is calculated incorrectly as 4000.00, which is 100 times too high.

A solution for the conflicting use of numbers to represent parameter values is to overload the `calculateInterest()` method. For example, in addition to the `calculateInterest()` method shown in Figure 4-12, you could add the method shown in Figure 4-13.

```
public static void calculateInterest(double bal, int rate)
{
    double interest, rateAsPercent;
    rateAsPercent = rate / 100.0;
    interest = bal * rateAsPercent;
    System.out.println("Simple interest on $" +
        bal + " at " + rate + "% rate is " +
        interest);
}
```

Notice the data type for rate.

Dividing by 100.0 converts rate to its percent equivalent.

Figure 4-13 The `calculateInterest()` method with a `double` parameter and an `int` parameter



In Figure 4-13, note that `rateAsPercent` 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`. Alternatively, you could use an explicit cast such as `rateAsPercent = (double)rate / 100`.

If an application calls the method `calculateInterest()` using two `double` arguments—for example, `calculateInterest(1000.00, 0.04)`—the first version of the method, the one shown in Figure 4-12, executes. However, if an integer is used as the second argument in a call to `calculateInterest()`—as in `calculateInterest(1000.00, 4)`—the second version of the method, the one shown in Figure 4-13, executes. In this second example, the whole number rate figure is correctly divided by `100.0` 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 interest figure—for example, `calculateInterestUsingDouble()` and `calculateInterestUsingInt()`. However, it is easier and more convenient for programmers who use your methods to remember just one method name they can use in the form that is most appropriate for their programs. It is convenient to be able to use one reasonable name for tasks that are functionally identical except for the argument types that can be passed to them. The compiler knows which method version to call based on the passed arguments.

Automatic Type Promotion in Method Calls

In Chapter 2, you learned that Java casts variables to a unifying type when you perform arithmetic with unlike types. For example, when you multiply an `int` and a `double`, the result is a `double`. In a similar way, Java can promote one data type to another when you pass a parameter to a method. For example, if a method has a `double` parameter and you pass in an integer, the integer is promoted to a `double`. Recall that the order of promotion is `double`, `float`, `long`, and `int`. Any type in this list can be promoted to any type that precedes it.

When an application contains just one version of a method, you can call the method using a parameter of the correct data type or one that can be promoted to the correct data type. For example, consider the simple method shown in Figure 4-14.

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

Figure 4-14 The `simpleMethod()` method with a `double` parameter

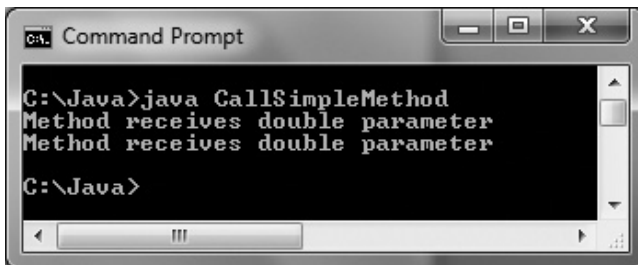
If you write an application in which you declare `doubleValue` as a `double` variable and `intValue` as an `int` variable (as shown in Figure 4-15), either of the two method calls,

`simpleMethod(doubleValue);` or `simpleMethod(intValue);`, results in the output “Method receives double parameter”. When you call the method with the `double` argument, the method works as expected, and when you call it with an integer argument, the integer is cast as (or promoted to) a `double`. The output of the program in Figure 4-15 is shown in Figure 4-16.

```
public class CallSimpleMethod
{
    public static void main(String[] args)
    {
        double doubleValue = 45.67;
        int intValue = 17;
        simpleMethod(doubleValue);
        simpleMethod(intValue);
    }
    public static void simpleMethod(double d)
    {
        System.out.println("Method receives double parameter");
    }
}
```

Either a double or an int can be sent to a method that accepts a double.

Figure 4-15 The `CallSimpleMethod` application that calls `simpleMethod()` with a `double` and an `int`



```

C:\Java>java CallSimpleMethod
Method receives double parameter
Method receives double parameter
C:\Java>
```

Figure 4-16 Output of the `CallSimpleMethod` application



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

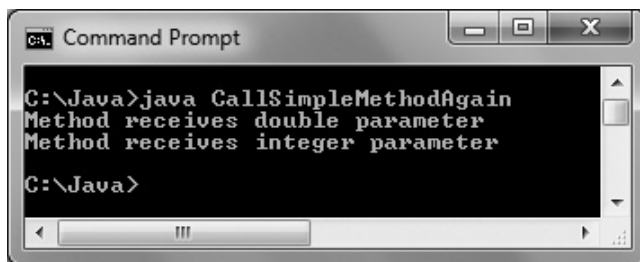
Suppose that you add an overloaded version of `simpleMethod()` to the program in Figure 4-15. This version accepts an integer parameter, as shown in Figure 4-17. When you properly overload a method, you can call it providing different argument lists, and

the appropriate version of the method executes. Now, the output changes when you call `simpleMethod(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”. Figure 4-18 shows the output.

```
public class CallSimpleMethodAgain
{
    public static void main(String[] args)
    {
        double doubleValue = 45.67;
        int intValue = 17;
        simpleMethod(doubleValue);
        simpleMethod(intValue);
    }
    public static void simpleMethod(double d)
    {
        System.out.println("Method receives double parameter");
    }
    public static void simpleMethod(int d)
    {
        System.out.println("Method receives integer parameter");
    }
}
```

The call with an `int` argument uses the method that is a better match when it is available.

Figure 4-17 The `CallSimpleMethodAgain` application that calls `simpleMethod()` with a `double` and an `int`



```
C:\Java>java CallSimpleMethodAgain
Method receives double parameter
Method receives integer parameter
C:\Java>
```

Figure 4-18 Output of the `CallSimpleMethodAgain` application

TWO TRUTHS & A LIE

Overloading a Method

1. When you overload Java methods, you write multiple methods with a shared name.
2. When you overload Java methods, the methods are called using different arguments.
3. Instead of overloading methods, it is preferable to write methods with unique identifiers.

193

The false statement is #3. Overloading methods is preferable to using unique identifiers because it is convenient for programmers to use one reasonable name for tasks that are functionally identical, except for the argument types that can be passed to them.



You Do It

Overloading Methods

In this section, you overload methods to display dates. The date-displaying methods might be used by many different applications in an organization, such as those that schedule jobs, appointments, and employee reviews. The methods take one, two, or three integer arguments. If there is one argument, it is the month, and the date becomes the first day of the given month in the year 2014. If there are two arguments, they are the month and the day in the year 2014. Three arguments represent the month, day, and year.



Instead of creating your own class to store dates, you can use the built-in Java class `GregorianCalendar` to handle dates. This exercise illustrates how some of the built-in `GregorianCalendar` class was constructed by Java's creators.

1. Open a new file in your text editor.
2. Begin the following `DemoOverLoad` class with three integer variables to test the method and three calls to a `displayDate()` method:

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

(continues)

(continued)

```
{
    int month = 6, day = 24, year = 2015;
    displayDate(month);
    displayDate(month, day);
    displayDate(month, day, year);
}
```

3. Create the following `displayDate()` method that requires one parameter to represent the month and uses default values for the day and year:

```
public static void displayDate(int mm)
{
    System.out.println("Event date " + mm + "/1/2014");
}
```

4. Create the following `displayDate()` method that requires two parameters to represent the month and day and uses a default value for the year:

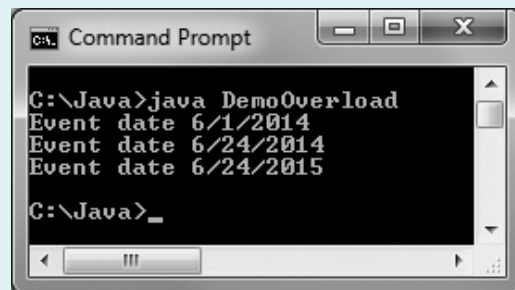
```
public static void displayDate(int mm, int dd)
{
    System.out.println("Event date " + mm + "/" + dd + "/2014");
}
```

5. Create the following `displayDate()` method that requires three parameters used as the month, day, and year:

```
public static void displayDate(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**.

8. Compile the program, correct any errors, recompile if necessary, and then execute the program. Figure 4-19 shows the output. Notice that whether you call the `displayDate()` method using one, two, or three arguments, the date is displayed correctly because you have successfully overloaded the `displayDate()` method.



```
C:\Java>java DemoOverload
Event date 6/1/2014
Event date 6/24/2014
Event date 6/24/2015
C:\Java>
```

Figure 4-19 Output of the `DemoOverload` application

Learning About Ambiguity

When you overload methods, you risk creating an **ambiguous** situation—one in which the compiler cannot determine which method to use. For example, consider the following overloaded `computeBalance()` method declarations:

```
public static void computeBalance(double deposit)
public static void computeBalance(double withdrawal)
```

If you declare a `double` variable named `myDeposit` and make a method call such as `computeBalance(myDeposit);`, you will have created an ambiguous situation. Both methods are exact matches for your call. You might argue that a call using a variable named `myDeposit` “seems” like it should go to the version of the method with the parameter named `deposit`, but Java makes no assumptions based on variable names. Each version of `computeBalance()` could accept a `double`, and Java does not presume which one you intended to use.

Sometimes, it is hard to recognize potentially ambiguous situations. For example, consider the following two method declarations:

```
public static void calculateInterest(int bal, double rate)
public static void calculateInterest(double bal, int rate)
```

These `calculateInterest()` methods have different types in their parameter lists. A call to `calculateInterest()` with an `int` and a `double` argument (in that order) executes the first version of the method, and a call to `calculateInterest()` with a `double` and an `int` 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 `calculateInterest()` using two integer arguments, as in `calculateInterest(300, 6);`, an ambiguous situation arises because there is no exact match for the method call. Because the two integers in the method call can be promoted to an integer and a `double` (thus matching the first version of the overloaded method), or to a `double` and an integer (thus matching the second version), the compiler does not know which version of the `calculateInterest()` method to use, and the program does not compile.

The two versions of `calculateInterest()` could coexist if no ambiguous calls were ever made. 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 parameter lists for methods with the same name. Methods with identical names that have identical parameter lists but different return types are not overloaded—they are illegal.

For example, the following two methods are illegal in the same class:

```
int aMethod(int x)
void aMethod(int x)
```

The compiler determines which of several versions of a method to call based on the arguments in the method call. If those two methods existed within a class, when the method



The compiler determines which version of a method to call by the method's signature. In Chapter 3, you learned that a method's signature is the combination of the method name and the number, types, and order of parameters.



If the keyword `final` appears in a method's parameter list, it is ignored when determining ambiguity. In other words, two methods with the headers `void aMethod(int x)` and `void aMethod(final int x)` are ambiguous.



Watch the video *Overloading Methods*.

TWO TRUTHS & A LIE

Learning About Ambiguity

1. When it is part of the same program as `void myMethod(int age, String name)`, the following method would be ambiguous:
`void myMethod(String name, int age)`
2. When it is part of the same program as `void myMethod(int age, String name)`, the following method would be ambiguous:
`String myMethod(int zipCode, String address)`
3. When it is part of the same program as `void myMethod(int age, String name)`, the following method would be ambiguous:
`void myMethod(int x, String y)`

The false statement is #1. A method that accepts an `int` parameter followed by a `String` is not ambiguous with one that accepts the parameters in the reverse order.

Creating and Calling Constructors with Parameters

In Chapter 3, you learned that Java automatically provides a constructor when you create a class. You also learned that you can write your own constructor, and that you often do so when you want to ensure that fields within classes are initialized to some appropriate default value. In Chapter 3, you learned that the automatically provided constructor is a default constructor (one that does not require arguments), and you learned that you can also write a custom default constructor. However, when you write your own constructors, you can also

write versions that receive parameters. Such parameters are often used to initialize data fields for an object.

For example, consider the `Employee` class with just one data field, shown in Figure 4-20. Its constructor assigns 999 to the `empNum` of each potentially instantiated `Employee` object. Anytime 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 ensure that the `partTimeWorker Employee`, like all `Employee` objects, will have an initial `empNum` of 999.

Alternatively, you might choose to create `Employee` objects with initial `empNum` values that differ for each `Employee`. To accomplish this when the object is instantiated, you can pass an employee number to the constructor. Figure 4-21 shows an `Employee` class that contains a constructor that receives a parameter. With this constructor, an argument is passed using a statement such as the following:

```
Employee partTimeWorker = new Employee(881);
```

When the constructor executes, the integer within the constructor call is passed to `Employee()` as the parameter `num`, which is assigned to the `empNum` field.

When you create an `Employee` class with a constructor such as the one shown in Figure 4-21, every `Employee` object you create must have an integer argument in its constructor call. In other words, with this new version of the class, the following statement no longer works:

```
Employee partTimeWorker = new Employee();
```

After you write a constructor for a class, you no longer receive the automatically provided default constructor. If a class's only constructor requires an argument, you must provide an argument for every object of the class that you create.

```
public class Employee
{
    private int empNum;
    Employee()
    {
        empNum = 999;
    }
}
```

Figure 4-20 The `Employee` class with a default constructor that initializes the `empNum` field

```
public class Employee
{
    private int empNum;
    Employee(int num)
    {
        empNum = num;
    }
}
```

Figure 4-21 The `Employee` class with a constructor that accepts a value

Overloading Constructors

As with any other method, you can overload constructors. Overloading constructors provides you with a way to create objects with different initializing arguments, or none, as needed. For example, in addition to using the provided constructor shown in Figure 4-21, you can create a

second constructor for the `Employee` class; Figure 4-22 shows an `Employee` class that contains two constructors. When you use this class to create an `Employee` object, you have the option of creating the object either with or without an initial `empNum` value. When you create an `Employee` object with the statement `Employee aWorker = new Employee();`, the constructor with no parameters 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 version that requires an integer is used, and the `anotherWorker Employee` receives an initial `empNum` of 7677.

You can use constructor 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. As long as the constructor parameter lists differ, the constructors are not ambiguous.

```
public class Employee
{
    private int empNum;
    Employee(int num)
    {
        empNum = num;
    }
    Employee()
    {
        empNum = 999;
    }
}
```

Figure 4-22 The `Employee` class that contains two constructors



Watch the video *Overloading Constructors*.

TWO TRUTHS & A LIE

Creating and Calling Constructors with Parameters

1. A default constructor is one that is automatically created.
2. When you write a constructor, it can be written to receive parameters or not.
3. If a class's only constructor requires an argument, you must provide an argument for every object of the class that you create.

The false statement is #1. A default constructor is one that takes no arguments. The constructor that is automatically created when you do not write your own version is a default constructor, but so is one that you write to take no arguments.



You Do It

Creating Overloaded Constructors

In this section, you create a class with overloaded constructors and demonstrate how they work.

1. Open a new file in your text editor, and start the `CarInsurancePolicy` class as follows. The class contains three fields that hold a policy number, the number of payments the policy holder will make annually, and the policy holder's city of residence.

```
public class CarInsurancePolicy
{
    private int policyNumber;
    private int numPayments;
    private String residentCity;
}
```

2. Create a constructor that requires parameters for all three data fields.

```
public CarInsurancePolicy(int num, int payments, String city)
{
    policyNumber = num;
    numPayments = payments;
    residentCity = city;
}
```

3. Suppose the agency that sells car insurance policies is in the city of Mayfield. Create a two-parameter constructor that requires only a policy number and number of payments. This constructor assigns *Mayfield* to `residentCity`.

```
public CarInsurancePolicy(int num, int payments)
{
    policyNumber = num;
    numPayments = payments;
    residentCity = "Mayfield";
}
```

4. Add a third constructor that requires only a policy number parameter. This constructor uses the default values of two annual payments and Mayfield as the resident city. (Later in this chapter, you will learn how to eliminate the duplicated assignments in these constructors.)

```
public CarInsurancePolicy(int num)
{
    policyNumber = num;
    numPayments = 2;
    residentCity = "Mayfield";
}
```

(continues)

(continued)

5. Add a `display()` method that outputs all the insurance policy data:

```
public void display()
{
    System.out.println("Policy #" + policyNumber + ". " +
        numPayments + " payments annually. Driver resides in " +
        residentCity + ".");
}
```

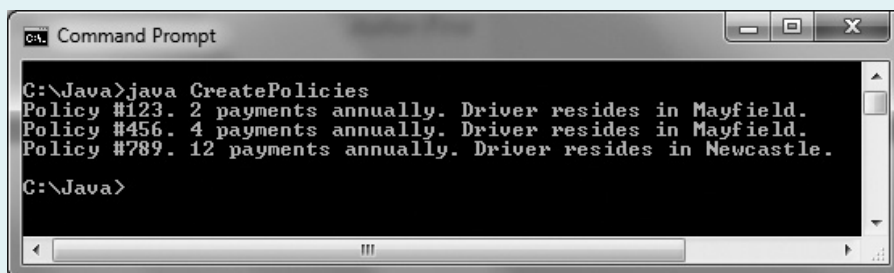
6. Add a closing curly brace for the class. Save the file as **CarInsurancePolicy.java**.
7. Open a new text file to create a short application that demonstrates the constructors at work. The application declares three `CarInsurancePolicy` objects using a different constructor version each time. Type the following code:

```
public class CreatePolicies
{
    public static void main(String[] args)
    {
        CarInsurancePolicy first = new CarInsurancePolicy(123);
        CarInsurancePolicy second = new CarInsurancePolicy(456, 4);
        CarInsurancePolicy third = new CarInsurancePolicy
            (789, 12, "Newcastle");
    }
}
```

8. Display each object, and add closing curly braces for the method and the class:

```
    first.display();
    second.display();
    third.display();
}
```

9. Save the file as **CreatePolicies.java**, and then compile and test the program. The output appears in Figure 4-23.



```
Command Prompt
C:\Java>java CreatePolicies
Policy #123. 2 payments annually. Driver resides in Mayfield.
Policy #456. 4 payments annually. Driver resides in Mayfield.
Policy #789. 12 payments annually. Driver resides in Newcastle.
C:\Java>
```

Figure 4-23 Output of the `CreatePolicies` program

(continues)

(continued)

10. Add a fourth declaration to the `CreatePolicies` class that attempts to create a `CarInsurancePolicy` object using a default constructor:

```
CarInsurancePolicy fourth = new CarInsurancePolicy();
```

11. Save and compile the revised `CreatePolicies` program. The class does not compile because the `CarInsurancePolicy` class does not contain a default constructor. Change the newly added declaration to a comment, compile the class again, and observe that the class now compiles correctly.

Examining Prewritten Overloaded Methods

In this section, you examine some built-in classes and recognize their correctly overloaded methods.

1. Using a Web browser, go to the Java Web site www.oracle.com/technetwork/java/index.html, and select **Java APIs** and **Java SE 7**.
2. Using the alphabetical list of classes, find the `PrintStream` class, and select it.
3. Examine the list of constructors for the class, and notice that each version has a unique parameter list.
4. Examine the list of methods named `print()` and `println()`. Notice that each overloaded version has a unique parameter list.
5. Using the alphabetical list of classes, find the `JOptionPane` class, and select it.
6. Examine the list of constructors for the class, and notice that each version has a unique parameter list.
7. Examine the list of methods named `showConfirmDialog()` and `showInputDialog()`. Notice that each overloaded version has a unique parameter list.

Learning About the `this` Reference

When you start creating classes, they can become large very quickly. Besides data fields, each class can have many methods, including several overloaded versions. On paper, a single class might require several pages of coded statements.

When you instantiate an object from a class, memory is reserved for each instance field in the class. For example, if a class contains 20 data fields, when you create one object from that class, enough memory is reserved to hold the 20 field values for that object. When you create 200 objects of the same class, the computer reserves enough memory for 4,000 data fields—20 fields for each of the 200 objects. In many applications, 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 needs a unique number, name, and salary value. (When you want each object to share a value, you define the field as `static`; a field is not `static` when the value in each object can be unique.) However, when you create a method for a class, any 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. Not only would it take an enormous amount of memory to store a separate copy of each method for every object created from a class, but memory would also be wasted because you would be storing identical copies of methods—that is, each object’s copy of the method would have the same contents. Luckily, in Java just one copy of each method in a class is stored, and all instantiated objects can use that copy.

When you use a nonstatic method, you use the object name, a dot, and the method name—for example, `aWorker.getEmpNum()` or `anotherWorker.getEmpNum()`. When you execute the `getEmpNum()` method, you are running the only copy of the method. However, within the `getEmpNum()` method, when you access the `empNum` field, you access a different field depending on the object. The compiler must determine *whose* copy of the `empNum` value should be returned by the single `getEmpNum()` method.

The compiler accesses the correct object’s field because every time you call a nonstatic method, you implicitly pass a reference to the named object attached to the method call. A **reference** is an object’s memory address. The reference is implicit because it is understood automatically without actually being written. The reference to an object that is passed to any object’s nonstatic method is called the **this reference**; `this` is a reserved word in Java. Only nonstatic, instance methods have a `this` reference. For example, the two `getEmpNum()` methods for the `Employee` class shown in Figure 4-24 perform identically. The first method simply uses the `this` reference without your being aware of it; the second method uses the `this` reference explicitly. Both methods return the `empNum` of the object used to call the method.

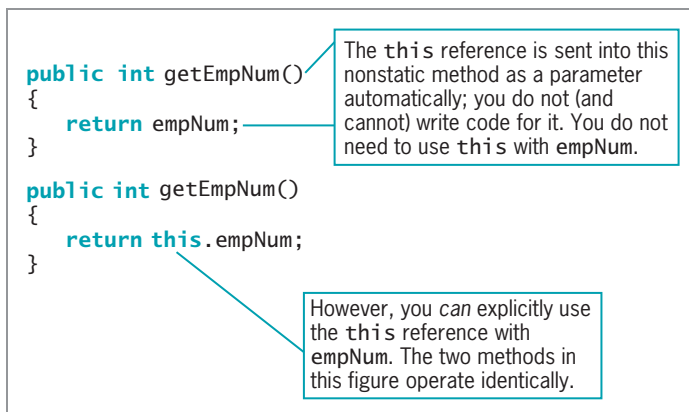


Figure 4-24 Two versions of the `getEmpNum()` method, with and without an explicit `this` reference

Frequently, you neither want nor need to refer to the `this` reference within the instance methods that 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.

On a few occasions, you must use the `this` reference to make your classes work correctly; one example is shown in the `Student` class in Figure 4-25. Within the constructor for this class, the parameter names `stuNum` and `gpa` are identical to the class field names. Within the constructor, `stuNum` and `gpa` refer to the locally declared names, not the class field names. The statement `stuNum = stuNum` accomplishes nothing—it assigns the local variable value to itself. The client application in Figure 4-26 attempts to create a `Student` object with an ID number of 111 and a grade point average of 3.5, but Figure 4-27 shows the incorrect output. The values are not assigned to the fields; instead, they are just zeroes.

```
public class Student
{
    private int stuNum;
    private double gpa;
    public Student (int stuNum, double gpa)
    {
        stuNum = stuNum;
        gpa = gpa;
    }
    public void showStudent()
    {
        System.out.println("Student #" + stuNum +
            " gpa is " + gpa);
    }
}
```

Don't Do It
All four variables used in these two statements are the local versions declared in the method's parameter list. The fields are never accessed because the local variables shadow the fields. These two assignment statements accomplish nothing.

Figure 4-25 A `Student` class whose constructor does not work

```
public class TestStudent
{
    public static void main(String[] args)
    {
        Student aPsychMajor =
            new Student(111, 3.5);
        aPsychMajor.showStudent();
    }
}
```

Figure 4-26 The `TestStudent` class that instantiates a `Student` object



```

C:\Java>java TestStudent
Student #0 gpa is 0.0
C:\Java>

```

Figure 4-27 Output of the TestStudent application using the incorrect Student class in Figure 4-25

One way to fix the problem with the Student class is to use different identifiers for the class's fields and the parameters to the constructor. However, sometimes the identifiers you have chosen are the best and simplest identifiers for a value. If you choose to use the same identifiers, you can use the `this` reference explicitly to identify the fields. Figure 4-28 shows a modified Student class. The only difference between this class and the one in Figure 4-25 is the explicit use of the `this` reference within the constructor. When the `this` reference is used with a field name in a class method, the reference is to the class field instead of to the local variable declared within the method. When the TestStudent application uses this new version of the Student class, the output appears as expected, as shown in Figure 4-29.

```

public class Student
{
    private int stuNum;
    private double gpa;
    public Student(int stuNum, double gpa)
    {
        this.stuNum = stuNum;
        this.gpa = gpa;
    }
    public void showStudent()
    {
        System.out.println("Student #" +
            stuNum + " gpa is " + gpa);
    }
}

```

Figure 4-28 The Student class using the explicit `this` reference within the constructor



```

C:\Java>java TestStudent
Student #111 gpa is 3.5
C:\Java>

```

Figure 4-29 Output of the TestStudent application using the new version of the Student class

Using the `this` Reference to Make Overloaded Constructors More Efficient

Suppose you create a `Student` class with data fields for a student number and a grade point average. Further suppose you want four overloaded constructors as follows:

- A constructor that accepts an `int` and a `double` and assigns them the student number and grade point average, respectively
- A constructor that accepts a `double` and assigns it to the grade point average, but initializes every student number to 999
- A constructor that accepts an `int` and assigns it to the student number, but initializes every grade point average to 0.0
- A default constructor that assigns 999 to every student number and 0.0 to every grade point average

Figure 4-30 shows the class. Although this class works, and allows `Students` to be constructed in four different ways, there is a lot of repetition within the constructors.

```
public class Student
{
    private int stuNum;
    private double gpa;
    Student(int num, double avg)
    {
        stuNum = num;
        gpa = avg;
    }
    Student(double avg)
    {
        stuNum = 999;
        gpa = avg;
    }
    Student(int num)
    {
        stuNum = num;
        gpa = 0.0;
    }
    Student()
    {
        stuNum = 999;
        gpa = 0.0;
    }
}
```

Figure 4-30 Student class with four constructors

You can reduce the amount of repeated code in Figure 4-30 and make the code less error-prone by calling one constructor version from the others. To do so, you use the `this` reference from one constructor version to call another version. Figure 4-31 shows how the `Student` class can be rewritten.

```
public class Student
{
    private int stuNum;
    private double gpa;
    Student(int num, double avg)
    {
        stuNum = num;
        gpa = avg;
    }
    Student(double avg)
    {
        this(999, avg);
    }
    Student(int num)
    {
        this(num, 0.0);
    }
    Student()
    {
        this(999, 0.0);
    }
}
```

Figure 4-31 The `Student` class using `this` in three of four constructors

By writing each constructor to call one master constructor, you save coding and reduce the chance for errors. For example, if code is added later to ensure that all student ID numbers are three digits, or that no grade point average is greater than 4.0, the new code will be written only in the two-parameter version of the constructor, and all the other versions will use it. (Testing a variable to ensure it falls within the proper range of values requires decision making. The next chapter covers this topic.)

Although you can use the `this` reference with field names in any method within a class, you cannot call `this()` from other methods in a class; you can only call it from constructors. Additionally, if you call `this()` from a constructor, it must be the first statement within the constructor.



Watch the video *The this Reference*.

TWO TRUTHS & A LIE

Learning About the `this` Reference

1. Usually, you want each instantiation of a class to have its own nonstatic data fields, but each object does not need its own copy of most methods.
2. When you use a nonstatic method, the compiler accesses the correct object's field because you implicitly pass an object reference to the method.
3. The `this` reference is supplied automatically in classes; you cannot use it explicitly.

The false statement is #3. Usually, you neither want nor need to refer to the `this` reference within the methods you write, but you can use it—for example, when there are conflicts between identifiers for fields and local variables.



You Do It

Using the `this` Reference to Make Constructors More Efficient

In this section, you modify the `CarInsurancePolicy` class so that its constructors are more efficient.

1. Open the **`CarInsurancePolicy.java`** file. Change the class name to **`CarInsurancePolicy2`**, and immediately save the file as **`CarInsurancePolicy2.java`**.
2. Change the name of the three-parameter constructor from `CarInsurancePolicy()` to **`CarInsurancePolicy2()`**.
3. Replace the constructor that accepts a single parameter for the policy number with the following constructor. The name of the constructor is changed from the earlier version, and this one passes the policy number and two constant values to the three-parameter constructor:

```
public CarInsurancePolicy2(int num)
{
    this(num, 2, "Mayfield");
}
```

(continues)

(continued)

4. Replace the constructor that accepts two parameters (for the policy number and number of payments) with the following constructor. This constructor has a new name and passes the two parameters and one constant value to the three-parameter constructor:

```
public CarInsurancePolicy2(int num, int payments)  
{  
    this(num, payments, "Mayfield");  
}
```

5. Save the file, and compile it.
6. Open the **CreatePolicies.java** file that demonstrates the use of the different constructor versions. Change the class name to **CreatePolicies2**, and save the file as **CreatePolicies2.java**.
7. Add the digit **2** in six places—three times to change the class name `CarInsurancePolicy` to **CarInsurancePolicy2** when the name is used as a data type, and in the three constructor calls.
8. Save the file, and then compile and execute it. The output is identical to that shown in Figure 4-23 in the previous “You Do It” section, but the repetitious constructor code has been eliminated.
9. You can further reduce the code in the `CarInsurancePolicy` class by changing the single-parameter constructor to the following, which removes the constant `"Mayfield"` from the constructor call:

```
public CarInsurancePolicy2(int num)  
{  
    this(num, 2);  
}
```

Now, the single-parameter version calls the two-parameter version and passes the policy number and the constant 2. In turn, the two-parameter version calls the three-parameter version, adding `"Mayfield"` as the city.

10. Save this version of the `CarInsurancePolicy2` class, and compile it. Then recompile the **CreatePolicies2.java** file, and execute it. The output remains the same.

Using static Fields

In Chapter 3, you learned that methods you create to use without objects are static. For example, the `main()` method in a program and the methods that `main()` calls without an object reference are static. You also learned that most methods you create within a class from which objects will be instantiated are nonstatic. Static 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. Whereas instance variables in a class exist separately for every object you create, there is only one copy of each **static** class variable per class. For example, consider the `BaseballPlayer` class in Figure 4-32. The `BaseballPlayer` class contains a shaded **static** field named `count`, and two nonstatic fields named `number` and `battingAverage`. The `BaseballPlayer` constructor sets values for `number` and `battingAverage` and increases the `count` by one. In other words, every time a `BaseballPlayer` object is constructed, it contains individual values for `number` and `battingAverage`, and the `count` field contains a count of the number of existing objects and is shared by all `BaseballPlayer` objects.

```
public class BaseballPlayer
{
    private static int count = 0;
    private int number;
    private double battingAverage;
    public BaseballPlayer(int id, double avg)
    {
        number = id;
        battingAverage = avg;
        count = count + 1;
    }
    public void showPlayer()
    {
        System.out.println("Player #" + number +
            " batting average is " + battingAverage +
            " There are " + count + " players");
    }
}
```

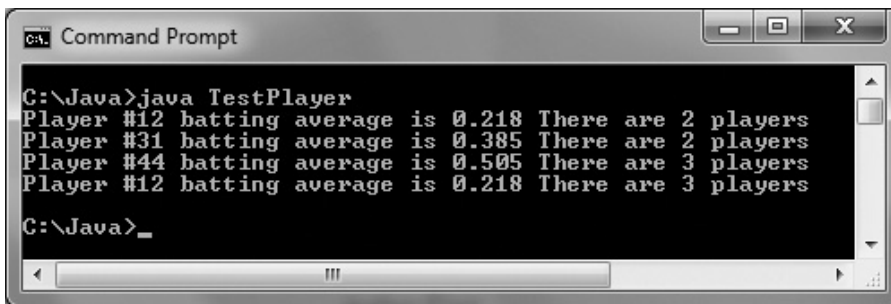
Figure 4-32 The `BaseballPlayer` class

The `showPlayer()` method in the `BaseballPlayer` class displays a `BaseballPlayer`'s number, batting average, and a count of all current players. This method is not **static**—it accesses an individual object's data. Methods declared as **static** cannot access instance variables, but instance methods can access both **static** and instance variables.

The `TestPlayer` class in Figure 4-33 is an application that declares two `BaseballPlayer` objects, displays them, and then creates a third `BaseballPlayer` object and displays it. When you examine the output in Figure 4-34, you can see that by the time the first two objects are declared, the `count` value that they share is 2. Whether `count` is accessed using the `aCatcher` object or the `aShortstop` object, the `count` is the same. After the third object is declared, its `count` value is 3, as is the value of `count` associated with the previously declared `aCatcher` object. In other words, because the **static** `count` variable is incremented within the class constructor, each object has access to the total number of objects that currently exist. No matter how many `BaseballPlayer` objects are eventually instantiated, each refers to the single `count` field.

```
public class TestPlayer
{
    public static void main(String[] args)
    {
        BaseballPlayer aCatcher = new BaseballPlayer(12, .218);
        BaseballPlayer aShortstop = new BaseballPlayer(31, .385);
        aCatcher.showPlayer();
        aShortstop.showPlayer();
        BaseballPlayer anOutfielder = new BaseballPlayer(44, .505);
        anOutfielder.showPlayer();
        aCatcher.showPlayer();
    }
}
```

Figure 4-33 The TestPlayer class



```
ca. Command Prompt
C:\Java>java TestPlayer
Player #12 batting average is 0.218 There are 2 players
Player #31 batting average is 0.385 There are 2 players
Player #44 batting average is 0.505 There are 3 players
Player #12 batting average is 0.218 There are 3 players
C:\Java>_
```

Figure 4-34 Output of the TestPlayer application

Using Constant Fields

In Chapter 2, you learned to create named constants by using the keyword `final`. Sometimes a data field in a class should be constant. For example, you might want to store a school ID value that is the same for every `Student` object you create, so you declare it to be static. In addition, if you want the value for the school ID to be fixed so that all `Student` objects use the same ID value—for example, when applying to scholarship-granting organizations or when registering for standardized tests—you might want to make the school ID unalterable. As with ordinary variables, you use the keyword `final` with a field to make its value unalterable after construction. For example, the class in Figure 4-35 contains the symbolic constant `SCHOOL_ID`. Because it is static, all objects share a single memory location for the field, and because it is `final`, it cannot change during program execution.


```
public class Student
{
    private static final int SCHOOL_ID = 12345;
    private int stuNum;
    private double gpa;
    public Student(int stuNum, double gpa)
    {
        this.stuNum = stuNum;
        this.gpa = gpa;
    }
    public void showStudent()
    {
        System.out.println("Student #" + stuNum +
            " gpa is " + gpa);
    }
}
```

Figure 4-35 The Student class containing a symbolic constant

A *nonstatic* `final` field's value can be set in the class constructor. For example, you can set it using a constant, or you can set it using a parameter passed into the constructor. However, a *static* `final` field's value must be set at declaration, as in the Student class example in Figure 4-35. This makes sense because there is only one *static* field stored for every object instantiated.



You can use the keyword `final` with methods or classes as well as with fields. When used in this manner, `final` indicates limitations placed on inheritance. You will learn more about inheritance in the chapters *Introduction to Inheritance* and *Advanced Inheritance Concepts*.



Fields that are `final` also can be initialized in a `static` initialization block. For more details about this technique, see the Java Web site.

Fields declared to be `static` are not always `final`. Conversely, `final` fields are not always `static`. In summary:

- If you want to create a field that all instantiations of the class can access but the field value can change, then it is `static` but not `final`. For example, in the last section you saw a nonfinal `static` field in the `BaseballPlayer` class that held a changing count of all instantiated objects.
- If you want each object created from a class to contain its own `final` value, you would declare the field to be `final` but not `static`. For example, you might want each `BaseballPlayer` object to have its own, nonchanging date of joining the team.
- If you want all objects to share a single nonchanging value, then the field is `static` and `final`.

TWO TRUTHS & A LIE

Using static Fields

1. Methods declared as `static` receive a `this` reference that contains a reference to the object associated with them.
2. Methods declared as `static` are called class methods.
3. A `final` `static` field's value is shared by every object of a class.

The false statement is #1. Static methods do not have a `this` reference because they have no object associated with them.



You Do It

Using Static and Nonstatic `final` Fields

In this section, you create a class for the Riverdale Kennel Club to demonstrate the use of static and nonstatic `final` fields. The club enters its dogs in an annual triathlon event in which each dog receives three scores in agility, conformation, and obedience.

1. Open a new file in your text editor, and enter the first few lines for a `DogTriathlonParticipant` class. The class contains a `final` field that holds the number of events in which the dog participated. Once a `final` field is set, it never should change. The field is not `static` because it is different for each dog. The class also contains a `static` field that holds the total cumulative score for all the participating dogs. The field is not `final` because its value increases as each dog participates in the triathlon, but it is `static` because at any moment in time, it is the same for all participants.

```
public class DogTriathlonParticipant
{
    private final int NUM_EVENTS;
    private static int totalCumulativeScore = 0;
```

2. Add six private fields that hold the participating dog's name, the dog's score in three events, the total score, and the average score:

```
private String name;
private int obedienceScore;
private int conformationScore;
private int agilityScore;
private int total;
private double avg;
```

(continues)

(continued)

- The constructor for the class requires five parameters—the dog's name, the number of events in which the dog participated, and the dog's scores in the three events. (After you read the chapter on decision making, you will be able to ensure that the number of nonzero scores entered matches the number of events, but for now no such checks will be made.) The constructor assigns each value to the appropriate field.

```
public DogTriathlonParticipant(String name,
    int numEvents, int score1, int score2, int score3)
{
    this.name = name;
    NUM_EVENTS = numEvents;
    obedienceScore = score1;
    conformationScore = score2;
    agilityScore = score3;
}
```

- After the assignments, the constructor calculates the total score for the participant and the participant's average score. Notice the result of the division is cast to a `double` so that any fractional part of the calculated average is not lost. Also, add the participant's total score to the cumulative score for all participants. Recall that this field is `static` because it should be the same for all participants at any point in time. After these statements, add a closing curly brace for the constructor.

```
total = obedienceScore +
    conformationScore + agilityScore;
avg = (double) total / NUM_EVENTS;
totalCumulativeScore = totalCumulativeScore +
    total;
}
```

- Start a method that displays the data for each triathlon participant.

```
public void display()
{
    System.out.println(name + " participated in " +
        NUM_EVENTS +
        " events and has an average score of " + avg);
    System.out.println("    " + name +
        " has a total score of " + total +
        " bringing the total cumulative score to " +
        totalCumulativeScore);
}
```

- Add a closing curly brace for the class. Then, save the file as **DogTriathlonParticipant.java**. Compile the class, and correct any errors.

(continues)

(continued)

7. Open a new file in your text editor, and then enter the header and opening and closing curly braces for a class you can use to test the `DogTriathlonParticipant` class. Also include a `main()` method header and its opening and closing braces.

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

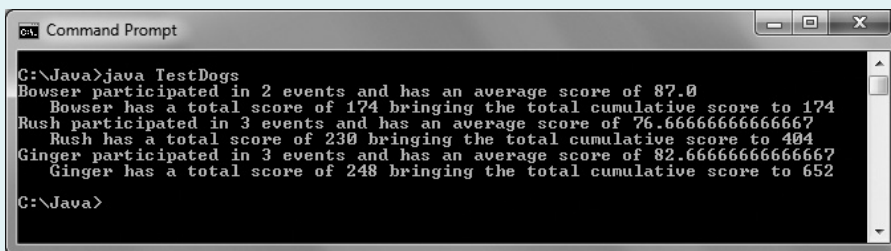
8. Between the braces of the `main()` method, declare a `DogTriathlonParticipant` object. Provide values for the participant's name, number of events, and three scores, and then display the object.

```
DogTriathlonParticipant dog1 =
    new DogTriathlonParticipant("Bowser", 2, 85, 89, 0);
dog1.display();
```

9. Create and display two more objects within the `main()` method.

```
DogTriathlonParticipant dog2 =
    new DogTriathlonParticipant("Rush", 3, 78, 72, 80);
dog2.display();
DogTriathlonParticipant dog3 =
    new DogTriathlonParticipant("Ginger", 3, 90, 86, 72);
dog3.display();
```

10. Save the file as **TestDogs.java**. Compile and execute the program. The output looks like Figure 4-36. Visually confirm that each total, average, and cumulative total is correct.



```
Command Prompt
C:\Java>java TestDogs
Bowser participated in 2 events and has an average score of 87.0
Bowser has a total score of 174 bringing the total cumulative score to 174
Rush participated in 3 events and has an average score of 76.66666666666667
Rush has a total score of 230 bringing the total cumulative score to 404
Ginger participated in 3 events and has an average score of 82.66666666666667
Ginger has a total score of 248 bringing the total cumulative score to 652
C:\Java>
```

Figure 4-36 Output of the `TestDogs` program

(continues)

(continued)

11. Experiment with the `DogTriathlonParticipant` class and its test class. For example, try the following:
 - Add a new statement at the end of the `TestDogs` class that again displays the data for any one of the participants. Note that as long as no new objects are created, the cumulative score for all participants remains the same no matter which participant uses it.
 - Try to assign a value to the `NUM_EVENTS` constant from the `display()` method, and then compile the class and read the error message generated.
 - Remove the keyword `static` from the definition of `totalCumulativeScore` in the `DogTriathlonParticipant` class, and then recompile the classes and run the program. Notice in the output that the nonstatic cumulative score no longer reflects the cumulative score for all objects but only the score for the current object using the `display()` method.
 - Use 0 as the number of events for an object. When the participant's average is calculated, the result is not numeric, and `NaN` is displayed. **NaN** is an acronym for *Not a Number*. In the next chapter, you will learn to make decisions, and then you can prevent the `NaN` output.

Using Automatically Imported, Prewritten Constants and Methods

Often, you need to create classes from which you will instantiate objects. You can create an `Employee` class with fields appropriate for describing employees in your organization and their functions, and an `Inventory` class with fields appropriate for whatever type of item you manufacture. However, many classes are commonly used by a wide variety of programmers. Rather than have each Java programmer “reinvent the wheel,” the creators of Java have produced hundreds of classes for you to use in your programs.

You have already used several of these predefined classes; for example, you have used the `System` and `JOptionPane` classes to produce output. Each of these classes is stored in a **package**, or a **library of classes**, which is simply a folder that provides a convenient grouping for classes. Many Java packages are available only if you explicitly name them within your program; for example, when you use `JOptionPane`, you must import the `javax.swing` package into your program. However, the group that contains classes such as `System` 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 **fundamental classes**, or basic classes, as opposed to the **optional classes** that must be named explicitly. Some references list a few other Java classes as also being “fundamental,” but the `java.lang` package is the only automatically imported, named package.

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. For example, a commonly used constant is `PI`. In geometry, π is an approximation of a circle's radius based on the ratio of the circumference of the circle to its diameter. Within the `Math` class, the declaration for `PI` is as follows:

```
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 and you can access it without declaring a `Math` object
- `double`, so it holds a floating-point value

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, you can calculate the area of a circle using the following statement:

```
areaOfCircle = java.lang.Math.PI * radius * radius;
```

However, the `java.lang` package is imported automatically into your programs, so if you simply reference `Math.PI`, Java recognizes this code as a shortcut to the full package path. Therefore, the preferred (and simpler) statement is the following:

```
areaOfCircle = Math.PI * radius * radius;
```

In addition to constants, many useful methods are 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. Table 4-1 lists some common `Math` class methods.

| Method | Value That the Method Returns |
|--------------------------|---|
| <code>abs(x)</code> | Absolute value of x |
| <code>acos(x)</code> | Arc cosine of x |
| <code>asin(x)</code> | Arc sine of x |
| <code>atan(x)</code> | Arc tangent of x |
| <code>atan2(x, y)</code> | Theta component of the polar coordinate (r , θ) that corresponds to the Cartesian coordinate x , y |
| <code>ceil(x)</code> | Smallest integral value not less than x (ceiling) |
| <code>cos(x)</code> | Cosine of x |

Table 4-1 Common `Math` class methods (*continues*)

(continued)

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

Table 4-1 Common Math class methods

Because all constants and methods in the `Math` class are classwide (that is, `static`), there is no need to create an instance of the `Math` class. 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 the square root of a negative number is undefined, if you display the result after the method call `imaginaryNumber = Math.sqrt(-12);`, you see `NaN`.

Importing Classes That Are Not Imported Automatically

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

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

In its `java.util` package, Java includes a `GregorianCalendar` class that is useful when working with dates and time. The Gregorian calendar is used in most of the world; it was instituted on October 15, 1582, and is named for Pope Gregory XIII, who was instrumental in the calendar's adoption. (For dates that fall before the Gregorian cutover date, the `GregorianCalendar` class uses the Julian calendar, which simply uses a different leap-year rule.)

You have seen examples in this book in which the `JOptionPane` and `Scanner` classes were imported so you could use their methods. Similarly, you can import the `java.util` class package, which includes the `GregorianCalendar` class.

You can instantiate an object of type `GregorianCalendar` from this class by using the full class path, as in the following:

```
java.util.GregorianCalendar myAnniversary = new java.util.GregorianCalendar();
```

Alternatively, when you include `import java.util.GregorianCalendar;` as the first line in your program, you can shorten the declaration of `myAnniversary` to this:

```
GregorianCalendar myAnniversary = new GregorianCalendar();
```

An alternative to importing a class is to import an entire package of classes. You can use the asterisk (`*`) as a **wildcard symbol**, which indicates that it can be replaced by any set of characters. In a Java `import` statement, you use a wildcard symbol to represent all the classes in a package. Therefore, the following statement imports the `GregorianCalendar` class and any other `java.util` classes as well:

```
import java.util.*;
```

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 use the data and method names that are part of the imported class or package. There is no disadvantage to importing an entire package instead of just the classes you need, and you will commonly see the wildcard method in professionally written Java programs. However, you have the alternative of importing each class you need individually. Importing each class by name, without wildcards, can be a form of documentation; this technique specifically shows which parts of the package are being used.

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.*;`. Also, note that the asterisk in an `import` statement imports all of the classes in a package, but not other packages that are within the imported package.



Your own classes are included in applications without `import` statements because of your `classpath` settings. See Appendix A for more information on `classpath`.

Using the `GregorianCalendar` Class

Seven constructors are available for `GregorianCalendar` objects. These constructors are overloaded, requiring different argument lists when they are called. The default constructor for the `GregorianCalendar` class creates a calendar object containing the current date and time in the default locale (time zone) that has been set for your computer. You can use other constructors to specify:

- year, month, date
- year, month, date, hour, minute
- year, month, date, hour, minute, second
- `Locale`
- `TimeZone`
- `TimeZone`, `Locale`

You can create a default `GregorianCalendar` object with a statement such as the following:

```
GregorianCalendar today = new  
GregorianCalendar();
```

Alternatively, you can create a `GregorianCalendar` object using one of the overloaded constructors—for example:

```
GregorianCalendar myGraduationDate = new  
GregorianCalendar(2014, 5, 24);
```

Specific data field values, such as the day, month, and year, can be retrieved from a `GregorianCalendar` object by using the `get()` method and specifying what you want as an argument. You could retrieve the day of the year (for example, February 1 is the 32nd day of the year) with the following statement:

```
int dayOfYear = today.get(GregorianCalendar.DAY_OF_YEAR);
```

The `GregorianCalendar` `get()` method always returns an integer. Some of the possible arguments to the `get()` method are shown in Table 4-2. In particular, notice that the month values in the `GregorianCalendar` class range from 0 through 11. Thus, January is month 0, February is month 1, and so on.

As an example of how to use a `GregorianCalendar` object, Figure 4-37 shows an `AgeCalculator` application. In this class, a default `GregorianCalendar` object named `now` is created. The user is prompted for a birth year, the current year is extracted from the `now` object using the `get()` method, and the user's age this year is calculated by subtracting the birth year from the current year. Figure 4-38 shows the output when a user born in 1986 runs the application in 2014.

| Arguments | Values Returned by get() |
|--------------|---|
| DAY_OF_YEAR | A value from 1 to 366 |
| DAY_OF_MONTH | A value from 1 to 31 |
| DAY_OF_WEEK | SUNDAY, MONDAY, ... SATURDAY, corresponding to values from 1 to 7 |
| YEAR | The current year; for example, 2012 |
| MONTH | JANUARY, FEBRUARY, ... DECEMBER, corresponding to values from 0 to 11 |
| HOUR | A value from 1 to 12; the current hour in the A.M. or P.M. |
| AM_PM | A.M. or P.M., which correspond to values from 0 to 1 |
| HOUR_OF_DAY | A value from 0 to 23 based on a 24-hour clock |
| MINUTE | The minute in the hour, a value from 0 to 59 |
| SECOND | The second in the minute, a value from 0 to 59 |
| MILLISECOND | The millisecond in the second, a value from 0 to 999 |

Table 4-2 Some possible arguments to and returns from the `GregorianCalendar` `get()` method

```
import java.util.*;
import javax.swing.*;
public class AgeCalculator
{
    public static void main(String[] args)
    {
        GregorianCalendar now = new GregorianCalendar();
        int nowYear;
        int birthYear;
        int yearsOld;
        birthYear = Integer.parseInt
            (JOptionPane.showInputDialog(null,
                "In what year were you born?"));
        nowYear = now.get(GregorianCalendar.YEAR);
        yearsOld = nowYear - birthYear;
        JOptionPane.showMessageDialog(null,
            "This is the year you become " + yearsOld +
            " years old");
    }
}
```

Figure 4-37 The `AgeCalculator` application



The `parseInt()` method is used in the application in Figure 4-37 to convert the user's input string to an integer. You learned about the `parseInt()` method in Chapter 2.



Figure 4-38 Execution of the AgeCalculator application

TWO TRUTHS & A LIE

Using Automatically Imported, Prewritten Constants and Methods

1. The creators of Java have produced hundreds of classes for you to use in your programs.
2. Java packages are available only if you explicitly name them within your program.
3. The implicitly imported `java.lang` package contains fundamental Java classes.

The false statement is #2. Many Java packages are available only if you explicitly name them within your program, but others are automatically imported.



You Do It

In this section, you learn more about using the `GregorianCalendar` class.

Using the Java Web Site

1. Using a Web browser, go to the Java Web site, and select **Java APIs** and **Java SE 7**. Using the alphabetical list of classes, find the `GregorianCalendar` class and select it.
2. Notice that `java.util` is cited at the top of the description, indicating that it is the containing package.
3. Read the history and background of the `GregorianCalendar` class to get an idea of how many issues are involved in determining values like the first day of the week and a week's number in a year. Then read the rest of the documentation to get a feel for the fields and methods that are available with the class.

(continues)

*(continued)**Using an Explicitly Imported, Prewritten Class*

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

1. Open a new file in your text editor. For the first line in the file, type the following:

```
import java.util.*;
```

2. On the next lines, begin the class by typing the class header, the opening brace, the `main()` method header, and its opening brace, as follows:

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

3. Declare a `GregorianCalendar` object named `now` that holds information about the current date and time. Then create a series of output statements that display a variety of `GregorianCalendar` fields containing information about the date:

```
GregorianCalendar now = new GregorianCalendar();  
System.out.println("YEAR: " + now.get(Calendar.YEAR));  
System.out.println("MONTH: " + now.get(Calendar.MONTH));  
System.out.println("WEEK_OF_YEAR: " +  
    now.get(Calendar.WEEK_OF_YEAR));  
System.out.println("WEEK_OF_MONTH: " +  
    now.get(Calendar.WEEK_OF_MONTH));  
System.out.println("DATE: " + now.get(Calendar.DATE));  
System.out.println("DAY_OF_MONTH: " +  
    now.get(Calendar.DAY_OF_MONTH));  
System.out.println("DAY_OF_YEAR: " +  
    now.get(Calendar.DAY_OF_YEAR));  
System.out.println("DAY_OF_WEEK: " +  
    now.get(Calendar.DAY_OF_WEEK));
```

4. Add more statements that display information about the current time, as follows:

```
System.out.println("AM_PM: " +  
    now.get(Calendar.AM_PM));  
System.out.println("HOUR: " + now.get(Calendar.HOUR));  
System.out.println("HOUR_OF_DAY: " +  
    now.get(Calendar.HOUR_OF_DAY));  
System.out.println("MINUTE: " +  
    now.get(Calendar.MINUTE));  
System.out.println("SECOND: " +  
    now.get(Calendar.SECOND));  
System.out.println("MILLISECOND: " +  
    now.get(Calendar.MILLISECOND));
```

5. Add the closing curly brace for the `main()` method and the closing curly brace for the class.

(continues)

(continued)

- Save the file as **CalendarDemo.java**. Compile and execute the program. Figure 4-39 shows the output from the program when it is executed a little before 10 a.m. on Thursday, April 3, 2014. Notice that although people usually think of April as month 4, the month in the output is 3—the month values in the `GregorianCalendar` are 0 through 11. When you display month values in your own programs, you might choose to add 1 to a value before displaying it, so that users see month numbers to which they are accustomed.

```

C:\Java>java CalendarDemo
YEAR: 2014
MONTH: 3
WEEK_OF_YEAR: 14
WEEK_OF_MONTH: 1
DATE: 3
DAY_OF_MONTH: 3
DAY_OF_YEAR: 93
DAY_OF_WEEK: 5
AM_PM: 0
HOUR: 9
HOUR_OF_DAY: 9
MINUTE: 56
SECOND: 41
MILLISECOND: 824
C:\Java>

```

Figure 4-39 Output of the `CalendarDemo` application

Creating an Interactive Application with a Timer

Next, you use the `GregorianCalendar` class to create an application that outputs a user's response time to a question.

- Open a new file in your text editor, and type the following two `import` statements. You need the `JOptionPane` class to use the `showConfirmDialog()` method, and you need the `java.util` package to use the `GregorianCalendar` class:

```

import javax.swing.JOptionPane;
import java.util.*;

```

- Begin the `DialogTimer` application as follows. Declare variables named `milli1`, `milli2`, `sec1`, and `sec2`. These are used to compute `time1` and `time2` from calendar objects created at the beginning and end of the program. You then use `time1` and `time2` to compute a `timeDifference`. Also declare a constant to hold the number of milliseconds in a second.

```

public class DialogTimer
{
    public static void main(String[] args)
    {
        int time1, time2, milli1, milli2, sec1,
            sec2, timeDifference;
        final int MILLISECS_IN_SECOND = 1000;

```

(continues)

(continued)

3. Instantiate a `GregorianCalendar` object, and retrieve its `MILLISECOND` and `SECOND` values. Compute a `time1` value by multiplying the current `sec1` value by 1000 and adding it to the current `milli1` value:

```
GregorianCalendar before = new GregorianCalendar();
milli1 = before.get(GregorianCalendar.MILLISECOND);
sec1 = before.get(GregorianCalendar.SECOND);
time1 = MILLISECS_IN_SECOND * sec1 + milli1;
```

4. Display a dialog box that asks the user to make a difficult choice:

```
JOptionPane.showConfirmDialog
    (null, "Is stealing ever justified? ");
```

5. Next, create a new `GregorianCalendar` object. This statement does not execute until the user provides a response for the dialog box, so the time variables contain different values from the first `GregorianCalendar` object created:

```
GregorianCalendar after = new GregorianCalendar();
milli2 = after.get(GregorianCalendar.MILLISECOND);
sec2 = after.get(GregorianCalendar.SECOND);
time2 = MILLISECS_IN_SECOND * sec2 + milli2;
```

6. Compute the difference between the times, and display the result in a dialog box.

```
timeDifference = time2 - time1;
JOptionPane.showMessageDialog(null, "It took " +
    timeDifference + " milliseconds for you to answer");
```

7. Add two closing curly braces—one for the method and the other for the class—and then save the file as **DialogTimer.java**.

8. Compile and execute the program. When the question appears, choose a response. The second output looks like Figure 4-40; the actual time displayed varies depending on how long you wait before selecting an answer.

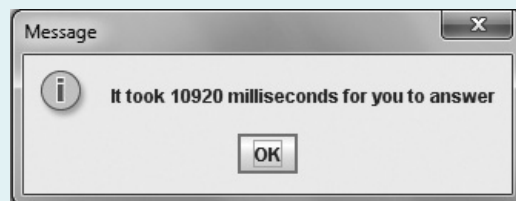


Figure 4-40 Output of the `DialogTimer` application

9. The output in the `DialogTimer` application is accurate only when the first and second `GregorianCalendar` objects are created during the same minute. For example, if the first object is created a few seconds before a new minute starts, and the second object is created a few seconds after the new minute starts, the second `SECOND` value appears to be much lower than the first one. On your own, modify the `DialogTimer` application to rectify this discrepancy. Save the file as **DialogTimer2.java**.

Understanding Composition and Nested Classes

Two of the ways that you can group classes are by using composition and by nesting classes. This section takes a brief look at both concepts.

Composition

The fields in a class can be simple data types like `int` and `double`, but they can also be class types. **Composition** describes the relationship between classes when an object of one class is a data field within another class. You have already studied many classes that contain `String` object fields. These classes employ composition.

When you use an object as a data member of another object, you must remember to supply values for the contained object if it has no default constructor. For example, you might create a class named `NameAndAddress` that stores name and address information. Such a class could be used for employees, customers, students, or anyone else who has a name and address. Figure 4-41 shows a `NameAndAddress` class. The class contains three fields, all of which are set by the constructor. A `display()` method displays the name and address information on three lines.

```
public class NameAndAddress
{
    private String name;
    private String address;
    private int zipCode;
    public NameAndAddress(String nm, String add, int zip)
    {
        name = nm;
        address = add;
        zipCode = zip;
    }
    public void display()
    {
        System.out.println(name);
        System.out.println(address);
        System.out.println(zipCode);
    }
}
```

Figure 4-41 The `NameAndAddress` class

Suppose you want to create a `School` class that holds information about a school. Instead of declaring fields for the `School`'s name and address, you could use the `NameAndAddress` class. The relationship created is sometimes called a **has-a relationship** because one class “has an” instance of another. Figure 4-42 shows a `School` class that declares and uses a `NameAndAddress` object.

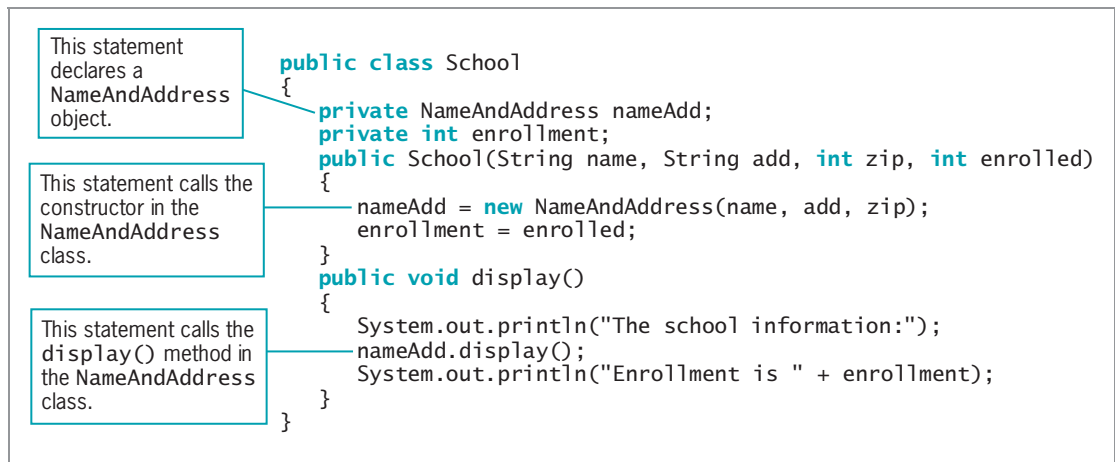


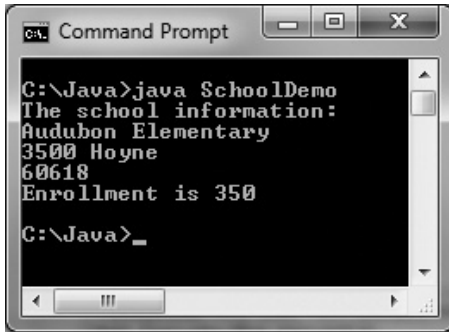
Figure 4-42 The School class

As Figure 4-42 shows, the `School` constructor requires four parameters. Within the constructor, three of the items—the name, address, and zip code—are passed to the `NameAndAddress` constructor to provide values for the appropriate fields. The fourth constructor parameter (the school's enrollment) is assigned to the `School` class enrollment field.

In the `School` class `display` method, the `NameAndAddress` object's `display()` method is called to display the school's name and address. The enrollment value is displayed afterward. Figure 4-43 shows a simple program that instantiates one `School` object. Figure 4-44 shows the execution.

```
public class SchoolDemo
{
    public static void main(String[] args)
    {
        School mySchool = new School
            ("Audubon Elementary",
            "3500 Hoyne", 60618, 350);
        mySchool.display();
    }
}
```

Figure 4-43 The SchoolDemo program



```
ca. Command Prompt
C:\Java>java SchoolDemo
The school information:
Audubon Elementary
3500 Hoyne
60618
Enrollment is 350

C:\Java>_
```

Figure 4-44 Output of the SchoolDemo program

Nested Classes

Every class you have studied so far has been stored in its own file, and the filename has always matched the class name. In Java, you can create a class within another class and store them together; such classes are **nested classes**. The containing class is the **top-level class**. There are four types of nested classes:

- **static member classes:** A static member class has access to all static methods of the top-level class.
- **Nonstatic member classes**, also known as **inner classes:** This type of class requires an instance; it has access to all data and methods of the top-level class.
- **Local classes:** These are local to a block of code.
- **Anonymous classes:** These are local classes that have no identifier.

The most common reason to nest a class inside another is because the inner class is used only by the top-level class; in other words, it is a “helper class” to the top-level class. Being able to package the classes together makes their connection easier to understand and their code easier to maintain.

For example, consider a `RealEstateListing` class used by a real estate company to describe houses that are available for sale. The class might contain separate fields for a listing number, the price, the street address, and the house’s living area. As an alternative, you might decide that although the listing number and price “go with” the real estate listing, the street address and living area really “go with” the house. So you might create an inner class like the one in Figure 4-45.

```
public class RealEstateListing
{
    private int listingNumber;
    private double price;
    private HouseData houseData;
    public RealEstateListing(int num, double price, String address,
        int sqFt)
    {
        listingNumber = num;
        this.price = price;
        houseData = new HouseData(address, sqFt);
    }
    public void display()
    {
        System.out.println("Listing number #" + listingNumber +
            " Selling for $" + price);
        System.out.println("Address: " + houseData.streetAddress);
        System.out.println(houseData.squareFeet + " square feet");
    }
    private class HouseData
    {
        private String streetAddress;
        private int squareFeet;
        public HouseData(String address, int sqFt)
        {
            streetAddress = address;
            squareFeet = sqFt;
        }
    }
}
```

Figure 4-45 The RealEstateListing class

Notice that the inner HouseData class in Figure 4-45 is a `private` class. You don't have to make an inner class `private`, but doing so keeps its members hidden from outside classes. If you wanted a class's members to be accessible, you would not make it an inner class. An inner class can access its top-level class's fields and methods, even if they are `private`, and an outer class can access its inner class's members.

You usually will not want to create inner classes. For example, if you made the HouseData class a regular class (as opposed to an inner class) and stored it in its own file, you could use it with composition in other classes—perhaps a MortgageLoan class or an Appraisal class. As it stands, it is usable only in the class in which it now resides. You probably will not create nested classes frequently, but you will see them implemented in some built-in Java classes.

TWO TRUTHS & A LIE

Understanding Composition and Nested Classes

1. Exposition describes the relationship between classes when an object of one class is a data field within another class.
2. When you use an object as a data member of another object, you must remember to supply values for the contained object if it has no default constructor.
3. A nested class resides within another class.

The false statement is #1. Composition describes the relationship between classes when an object of one class is a data field within another class.

Don't Do It

- Don't try to use a variable that is out of scope.
- Don't assume that a constant is still a constant when passed to a method's parameter. If you want a parameter to be constant within a method, you must use `final` in the parameter list.
- Don't try to overload methods by giving them different return types. If their identifiers and parameter lists are the same, then two methods are ambiguous no matter what their return types are.
- Don't think that *default constructor* means only the automatically supplied version. A constructor with no parameters is a default constructor, whether it is the one that is automatically supplied or one you write.
- Don't forget to write a default constructor for a class that has other constructors if you want to be able to instantiate objects without using arguments.

Key Terms

A **block** is the code between a pair of curly braces.

An **outside block**, or **outer block**, contains another block.

An **inside block**, or **inner block**, is contained within another block.

Nested describes the state of an inner block.

Comes into scope describes what happens to a variable when it becomes usable.

Goes out of scope describes what happens to a variable when it ceases to exist at the end of the block in which it is declared.

To **redeclare a variable** is to attempt to declare it twice—an illegal action.

A variable **overrides** another with the same name when it takes precedence over the other variable.

Shadowing is the action that occurs when a local variable hides a variable with the same name that is further away in scope.

Closer in scope describes the status of a local variable over others that it shadows.

Overloading involves using one term to indicate diverse meanings, or writing multiple methods with the same name but with different arguments.

An **ambiguous** situation is one in which the compiler cannot determine which method to use.

A **reference** is an object's memory address.

The **this reference** is a reference to an object that is passed to any object's nonstatic class method.

Class methods are static methods that do not have a `this` reference (because they have no object associated with them).

Class variables are static variables that are shared by every instantiation of a class.

NaN is an acronym for *Not a Number*.

A **package** is a library of classes.

A **library of classes** is a folder that provides a convenient grouping for classes.

The `java.lang` package is implicitly imported into every Java program.

The **fundamental classes** are basic classes contained in the `java.lang` package that are automatically imported into every program you write.

The **optional classes** reside in packages that must be explicitly imported into your programs.

A **wildcard symbol** is an asterisk—a symbol used to indicate that it can be replaced by any set of characters. In a Java `import` statement, you use a wildcard symbol to represent all the classes in a package.

Composition describes the relationship between classes when an object of one class is a data field within another class.

A **has-a relationship** is a relationship based on composition.

Nested classes are classes contained in other classes.

The **top-level class** is the containing class in nested classes.

A **static member class** is a type of nested class that has access to all `static` methods of its top-level class.

Nonstatic member classes, also known as **inner classes**, are nested classes that require an instance.

Local classes are a type of nested class that are local to a block of code.

Anonymous classes are nested, local classes that have no identifier.

Chapter Summary

- A variable's scope is the portion of a program within which you can reference that variable. A block is the code between a pair of curly braces. 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. If you declare a variable within a class and use the same variable name within a method of the class, the variable used inside the method takes precedence over (or overrides, or masks) the first variable.
- Overloading involves writing multiple methods with the same name but different parameter lists. Methods that have identical parameter lists but different return types are not overloaded; they are illegal.
- When you overload methods, you risk creating an ambiguous situation—one in which the compiler cannot determine which method to use.
- When you write your own constructors, they can receive parameters. Such parameters are often used to initialize data fields for an object. After you write a constructor for a class, you no longer receive the automatically provided default constructor. If a class's only constructor requires an argument, you must provide an argument for every object of the class that you create. You can overload constructors just as you can other methods.
- Within nonstatic methods, data fields for the correct object are accessed because a `this` reference is implicitly passed to nonstatic 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 fields and methods are shared by every instantiation of a class. When a field in a class is `final`, it cannot change after it is assigned its initial value.
- Java contains hundreds of prewritten classes that are stored in packages, which are folders that provide convenient groupings 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. The `GregorianCalendar` class allows you to define and manipulate dates and time.

- Composition describes the relationship between classes when an object of one class is a data field within another class. You can create nested classes that are stored in the same file. The most common reason to nest a class inside another is because the inner class is used only by the outer or top-level class; in other words, it is a “helper class” to the top-level class.

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 variables with the same name multiple times _____.
 - a. within a statement
 - b. within a block
 - c. within a method
 - d. You never can declare multiple variables with the same name.
5. If you declare a variable as an instance variable within a class, and you declare and use the same variable name within a method of the class, then within the method, _____.
 - a. the variable used inside the method takes precedence
 - b. the class instance variable takes precedence
 - c. the two variables refer to a single memory address
 - d. an error will occur
6. A method variable _____ a class variable with the same name.
 - a. acquiesces to
 - b. destroys
 - c. overrides
 - d. alters

7. Nonambiguous, overloaded methods must have the same _____.
- a. name
 - b. number of parameters
 - c. parameter names
 - d. types of parameters
8. If a method is written to receive a `double` parameter, 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 _____ parameters.
- 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 that contains one method and instantiate two objects, you usually store _____ for use with the objects.
- a. one copy of the method
 - b. two copies of the method
 - c. two different methods containing two different `this` references
 - d. data only (the methods are not stored)
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 reference 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` used with 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 seven constructors. This means that the constructors _____.
 - a. override each other
 - b. are ambiguous
 - c. are overloaded
 - d. all of the above

20. The `GregorianCalendar` class `get()` method always returns a(n) _____.
 - a. day of the week
 - b. date
 - c. integer
 - d. `GregorianCalendar` object

Exercises



Programming Exercises

1. Create a class named `FormLetterWriter` that includes two overloaded methods named `displaySalutation()`. The first method takes one `String` parameter that represents a customer's last name, and it displays the salutation "Dear Mr. or Ms." followed by the last name. The second method accepts two `String` parameters that represent a first and last name, and it displays the greeting "Dear" followed by the first name, a space, and the last name. After each salutation, display the rest of a short business letter: "Thank you for your recent order." Write a `main()` method that tests each overloaded method. Save the file as **FormLetterWriter.java**.

2. Create a class named `Billing` that includes three overloaded `computeBill()` methods for a photo book store.
 - When `computeBill()` receives a single parameter, it represents the price of one photo book ordered. Add 8% tax, and return the total due.
 - When `computeBill()` receives two parameters, they represent the price of a photo book and the quantity ordered. Multiply the two values, add 8% tax, and return the total due.
 - When `computeBill()` receives three parameters, they represent the price of a photo book, the quantity ordered, and a coupon value. Multiply the quantity and price, reduce the result by the coupon value, and then add 8% tax and return the total due.

Write a `main()` method that tests all three overloaded methods. Save the application as **Billing.java**.

3.
 - a. Create a `BirdSighting` class for the Birmingham Birdwatcher's Club that includes data fields for a bird species sighted, the number seen, and the day of the year. For example, April 1 is the 91st day of the year, assuming it is not a leap year. The class also includes methods to get each field. In addition, create a default constructor that automatically sets the species to "robin" and the number and day to 1. Save the file as **BirdSighting.java**. Create an application named `TestBirdSighting` that demonstrates that each method works correctly. Save the file as **TestBirdSighting.java**.
 - b. Create an additional overloaded constructor for the `BirdSighting` class you created in Exercise 3a. This constructor receives parameters for each of the data fields and assigns them appropriately. Add any needed statements to the `TestBirdSighting` application to ensure that the overloaded constructor works correctly, save it, and then test it.
 - c. Create a class with the same functionality as the `BirdSighting` class, but create the default constructor to call the three-parameter constructor. Save the class as **BirdSighting2.java**. Create an application to test the new version of the class, and name it **TestBirdSighting2.java**.
4.
 - a. Create a class named `BloodData` that includes fields that hold a blood type (the four blood types are *O*, *A*, *B*, and *AB*) and an Rh factor (the factors are + and -). Create a default constructor that sets the fields to "O" and "+", and an overloaded constructor that requires values for both fields. Include get and set methods for each field. Save this file as **BloodData.java**. Create an application named `TestBloodData` that demonstrates that each method works correctly. Save the application as **TestBloodData.java**.
 - b. Create a class named `Patient` that includes an ID number, age, and `BloodData`. Provide a default constructor that sets the ID number to "0", the age to 0, and the `BloodData` to "O" and "+". Create an overloaded constructor that provides values for each field. Also provide get methods for each field. Save the file as **Patient.java**. Create an application named `TestPatient` that demonstrates that each method works correctly, and save it as **TestPatient.java**.

5.
 - a. Create a class for the Tip Top Bakery named `Bread` with data fields for bread type (such as “rye”) and calories per slice. Include a constructor that takes parameters for each field, and include get methods that return the values of the fields. Also include a `public final static String` named `MOTTO` and initialize it to *The staff of life*. Write an application named `TestBread` to instantiate three `Bread` objects with different values, and then display all the data, including the motto, for each object. Save both the **`Bread.java`** and **`TestBread.java`** files.
 - b. Create a class named `SandwichFilling`. Include a field for the filling type (such as “egg salad”) and another for the calories in a serving. Include a constructor that takes parameters for each field, and include get methods that return the values of the fields. Write an application named `TestSandwichFilling` to instantiate three `SandwichFilling` objects with different values, and then display all the data for each object. Save both the **`SandwichFilling.java`** and **`TestSandwichFilling.java`** files.
 - c. Create a class named `Sandwich`. Include a `Bread` field and a `SandwichFilling` field. Include a constructor that takes parameters for each field needed in the two objects and assigns them to each object’s constructor. Write an application named `TestSandwich` to instantiate three `Sandwich` objects with different values, and then display all the data for each object, including the total calories in a `Sandwich`, assuming that each `Sandwich` is made using two slices of `Bread`. Save both the **`Sandwich.java`** and **`TestSandwich.java`** files.
6.
 - a. Create a class named `Circle` with fields named `radius`, `diameter`, and `area`. Include a constructor that sets the radius to 1 and calculates the other two values. Also include methods named `setRadius()` and `getRadius()`. The `setRadius()` method not only sets the radius but also calculates the other two values. (The diameter of a circle is twice the radius, and the area of a circle is π multiplied by the square of the radius. Use the `Math` class `PI` constant for this calculation.) Save the class as **`Circle.java`**.
 - b. Create a class named `TestCircle` whose `main()` method declares several `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. Display all the values for all the `Circle` objects. Save the application as **`TestCircle.java`**.
7. Write a Java application that uses the `Math` class to determine the answers for each of the following:
 - a. The square root of 37
 - b. The sine and cosine of 300
 - c. The value of the floor, ceiling, and round of 22.8

- d. The larger and the smaller of the character “D” and the integer 71
- e. A random number between 0 and 20 (*Hint:* The `random()` method returns a value between 0 and 1; you want a number that is 20 times larger.)

Save the application as **MathTest.java**.

8. Write an application that uses methods in the `GregorianCalendar` class to calculate how many days are left until the first day of next month. Save the file as **NextMonth.java**.
9. Write an application that uses methods in the `GregorianCalendar` class to calculate the number of days from today until the end of the current year. Save the file as **YearEnd.java**.
10.
 - a. Create a `CertificateOfDeposit` class. The class contains data fields that hold a certificate number, account holder’s last name, balance, issue date, and maturity date, using `GregorianCalendar` objects for each date. Provide get and set methods for each field. Also provide a constructor that requires parameters used to set the first four fields, and sets the maturity date to exactly one year after the issue date. Save the class as **CertificateOfDeposit.java**.
 - b. Create an interactive application that prompts the user for data for two `CertificateOfDeposit` objects. Prompt the user for certificate number, name, balance, and issue date for each `CertificateOfDeposit`, and then instantiate the objects. Display all the values, including the maturity dates. Save the application as **TestCertificateOfDeposit.java**.
11. Create a class named `State` that holds the following fields: a `String` for the name of the state, an integer for the population, and two `City` objects that hold data about the capital city and the most populous city. The `State` constructor requires six parameters that represent the names and populations of the state, its capital, and its most populous city. Provide get methods for each field. Create the `City` class to be a nonstatic, private inner class within the `State` class; the `City` class contains a city’s name and population. Create a class to assign values to and display values from two `State` objects. Save the files as **State.java** and **TestState.java**.



Debugging Exercises

1. Each of the following files in the `Chapter04` folder of your downloadable student files has syntax and/or logic 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 `DebugFour1.java` as `FixDebugFour1.java`.
 - a. `DebugFour1.java`
 - b. `DebugFour2.java`
 - c. `DebugFour3.java` and `DebugBox.java`
 - d. `DebugFour4.java`



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



Game Zone

1. Dice are used in many games. One die can be thrown to randomly show a value from 1 through 6. Design a `Die` class that can hold an integer data field for a value (from 1 to 6). Include a constructor that randomly assigns a value to a die object. Appendix D contains information on generating random numbers. To fully understand the process, you must learn more about Java classes and methods. However, for now, you can copy the following statement to generate a random number between 1 and 6 and assign it to a variable. Using this statement assumes you have assigned appropriate values to the static constants.

```
randomValue = ((int)(Math.random() * 100) % HIGHEST_DIE_VALUE +  
LOWEST_DIE_VALUE);
```

Also include a method in the class to return a die's value. Save the class as **Die.java**.

Write an application that randomly “throws” two dice and displays their values. After you read the chapter *Making Decisions*, you will be able to have the game determine the higher die. For now, just observe how the values change as you execute the program multiple times. Save the application as **TwoDice.java**.

2. Using the `Die` class, write an application that randomly “throws” five dice for the computer and five dice for the player. Display the values and then, by observing the results, decide who wins based on the following hierarchy of `Die` values. (The computer will not decide the winner; the player will determine the winner based on observation.) Any higher combination beats a lower one; for example, five of a kind beats four of a kind.
 - Five of a kind
 - Four of a kind
 - Three of a kind
 - A pair

After you learn about decision making in the next chapter, you will be able to make the program determine whether you or the computer had the better roll, and after you read the chapter *Arrays*, you will be able to make the determination more efficient. For now, just observe how the values change as you execute the program multiple times. Save the application as **FiveDice.java**.



Case Problems

These projects build on the ones you created in Chapter 3, so they have the same filenames. If you want to retain both versions of the files, save them in different folders.

239

1. a. Carly's Catering provides meals for parties and special events. In Chapter 3, you created an `Event` class for the company. The `Event` class contains two `public final static` fields that hold the price per guest (\$35) and the cutoff value for a large event (50 guests), and three `private` fields that hold an event number, number of guests for the event, and the price. It also contains two `public set` methods and three `public get` methods.

Now, modify the `Event` class to contain two overloaded constructors.

- One constructor accepts an event number and number of guests as parameters. Pass these values to the `setEventNumber()` and `setGuests()` methods, respectively. The `setGuests()` method will automatically calculate the event price.
- The other constructor is a default constructor that passes "A000" and 0 to the two-parameter constructor.

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

- b. In Chapter 3, you also created an `EventDemo` class to demonstrate using two `Event` objects. Now, modify that class to instantiate two `Event` objects, and include the following new methods in the class:

- Instantiate one object to retain the constructor default values.
- Accept user data for the event number and guests fields, and use this data set to instantiate the second object. Display all the details for both objects.

Save the file as **EventDemo.java**.

2. a. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. In Chapter 3, you created a `Rental` class for the company. The `Rental` class contains two `public final static` fields that hold the number of minutes in an hour and the hourly rental rate (\$40), and four `private` fields that hold a contract number, number of hours for the rental, number of minutes over an hour, and the price. It also contains two `public set` methods and four `public get` methods.

Now, modify the `Rental` class to contain two overloaded constructors.

- One constructor accepts a contract number and number of minutes as parameters. Pass these values to the `setContractNumber()` and `setHoursAndMinutes()` methods, respectively. The `setHoursAndMinutes()` method will automatically calculate the hours, extra minutes, and price.

- The other constructor is a default constructor that passes “A000” and 0 to the two-parameter constructor.

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

b. In Chapter 3, you also created a `RentalDemo` class to demonstrate a `Rental` object. Now, modify that class to instantiate two `Rental` objects.

- Instantiate one object to retain the constructor default values.
- Accept user data for the contract number and minutes fields and use this data set to instantiate the second object. Display all the details for both objects.

Save the file as **RentalDemo.java**.