Java Language Companion for

Starting Out with Programming Logic and Design, 3rd Edition

By Tony Gaddis

Table of Contents

Introduction

Welcome to the Java Language Companion for *Starting Out with Programming Logic and Design, 2nd
Edition*, by Tony Gaddis. You can use this guide as a reference for the Java Programming Language as
you work through the textbook. Each chapter in this guide corresponds to the same numbered chapter
in the textbook. As you work through a chapter in the textbook, you can refer to the corresponding
chapter in this guide to see how the chapter's topics are implemented in the Java programming
language. In this book you will also find Java versions of many of the pseudocode programs that are
presented in the textbook.

Chapter 1   Introduction to Computers and Programming

About the Java Compiler and the Java Virtual Machine

When a Java program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The Java programming statements written by the programmer are called *source code*, and the file they are saved in is called a *source file*. Java source files end with the *.java* extension.

After the programmer saves the source code to a file, he or she runs the Java compiler. A *compiler* is a program that translates source code into an executable form. During the translation process, the compiler uncovers any syntax errors that may be in the program. *Syntax errors* are mistakes that the programmer has made that violate the rules of the programming language. These errors must be corrected before the compiler can translate the source code. Once the program is free of syntax errors, the compiler creates another file that holds the translated instructions.

Most programming language compilers translate source code directly into files that contain machine language instructions. These files are called *executable files* because they may be executed directly by the computer's CPU. The Java compiler, however, translates a Java source file into a file that contains byte code instructions. Byte code instructions are not machine language, and therefore cannot be directly executed by the CPU. Instead, they are executed by the Java Virtual Machine. The *Java Virtual Machine* (JVM) is a program that reads Java byte code instructions and executes them as they are read. For this reason, the JVM is often called an interpreter, and Java is often referred to as an interpreted language.

Although Java byte code is not machine language for a CPU, it can be considered as machine language for the JVM. You can think of the JVM as a program that simulates a computer whose machine language is Java byte code.

The Java Development Kit

To write Java programs you need have the Java Development Kit (JDK) installed on your computer. It is probably already installed in your school's computer lab. On your own computer, you can download the JDK from the following Web site:

http://java.sun.com/javase/downloads

This Web site provides several different bundles of software that you can download. You simply need to download the latest version of the Java SE Development kit. (If you plan to work through Chapter 15 of your textbook, you will probably want to download the bundle that includes the JDK and NetBeans.)

There are many different development environments available for Java, and your instructor probably has his or her favorite one. It's possible that your instructor will require you to download and install other software, in addition to the Java JDK. If this is the case, your instructor will probably provide instructions for using that development environment. If you are not using a particular development environment in your class, the following tutorial takes you through the steps for writing a Java program using a plain text editor, then compiling and executing the program using the JDK command line utilities.

Note -- In the following tutorial you will be working at the operating system command prompt. To complete the tutorial you need to know how to open a command prompt window, and how to use an operating system command to change your working directory (folder).

Tutorial: Compiling and Running a Java Program using the JDK Command Line Utilities

STEP 1: First you will write a simple Java program. Open NotePad (if you are using Windows) or any other plain text editor that you choose.

STEP 2: Type the following Java program exactly as it appears here:

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World");
   }
}
```

As you type the program, make sure that you match the case of each character. If you do not type the program exactly as it is shown here, an error will probably occur later.

STEP 3: Save the file as `HelloWorld.java`. (Once again, make sure the case of each character in the file name matches that shown here.) You can save the file in any folder that you choose. Just make sure you remember where you saved it, as you will be going back to that folder in a moment.

STEP 4: Open a command prompt window on your system. Change your current working directory to the folder where you saved the `HelloWorld.java` program in Step 3.

STEP 5: At the command prompt, type the following command exactly as it is shown,and press Enter:

```
javac HelloWorld.java
```

This command compiles the program that you wrote earlier. If you typed the program exactly as it was shown in Step 2, you should not see any error messages. If you do see error messages, open the file in the text editor and correct any typing mistakes that you made, save the file, and repeat this step. If you did not see any error messages, continue with Step 6.

STEP 6: Type the following command, exactly as it appears, to run the program:

```
java HelloWorld
```

When the program runs, you should see the message Hello World displayed on the screen.

Chapter 2   Input, Processing, and Output

Setting Up a Java Program

When you start a new Java program you must first write a *class declaration* to contain your Java code. Class declarations are very important in the Java language, and when you progress to more advanced programming techniques you will learn a great deal more about them. For now, simply think of a class declaration as a container for Java code. Here is an example

```
public class Simple
{

}
```

The first line of the class declaration is called the *class header*. In this example the class header reads:

```
    public class Simple
```

The words `public` and `class` are key words in the Java language, and the word `Simple` is the name of the class. When you write a class declaration you have to give the class a name. Notice that the words `public` and `class` are written in all lowercase letters. In Java, all key words are written in lowercase letters. If you mistakenly write an uppercase letter in a key word, an error will occur when you compile the program.

The class name, which in this case is `Simple`, does not have to be written in all lowercase letters because it is not a key word in the Java language. This is just a name that I made up when I wrote the class declaration. Notice that I wrote the first character of the class name in uppercase. It is not required, but it is a standard practice in Java to write the first character of a class name in uppercase. Java programmers do this so class names are more easily distinguishable from the names of other items in a program.

Another important thing to remember about the class name is that it must be the same as the name of the file that the class is stored in. For example, if I create a class named `Simple` (as shown previously), that class declaration will be stored in a file named `Simple.java`. (All Java source code files must be named with the `.java` extension.)

Notice that a set of curly brace follows the class header. Curly braces are meant to enclose things, and these curly braces will enclose all of the code that will be written inside the class. So, the next step is to write some code inside the curly braces.

Inside the class's curly braces you must write the definition of a method named `main`. A *method* is another type of container that holds code. When a Java program executes, it automatically begins running the code that is inside the `main` method. Here is how my `Simple` class will appear after I've added the `main` method declaration:

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

    }
}
```

The first line of the method definition, which is called the *method header*, begins with the words `public static void main` and so forth. At this point you don't need to be concerned about what any of these words mean. Just remember that you have to write the method header *exactly* as it is shown. Notice that a set of curly braces follow the method header. All of the code that you will write inside the method must be written inside these curly braces.

Displaying Screen Output

To display text on the screen in Java you use the following statements:

- `System.out.println()`
- `System.out.print()`

First, let's look at the `System.out.println()` statement. The purpose of this statement is to display a line of output. Notice that the statement ends with a set of parentheses. The text that you want to display is written as a string inside the parentheses. Program 2-1 shows an example. (This is the Java version of pseudocode Program 2-1 in your textbook.)

First, a note about the line numbers that you see in the program. These are *NOT* part of the program! They are helpful, though, when we need to discuss parts of the code. We can simply refer to specific line numbers and explain what's happening in those lines. For that reason we will show line numbers in all of our program listings. When you are writing a program, however, do not type line numbers in your code. Doing so will cause a mountain of errors when you compile the program!

Program 2-1 has three `System.out.println()` statements, appearing in lines 5, 6, and 7. (I told you those line numbers would be useful!) Line 5 displays the text `Kate Austen`, line 6 displays the text `123 Dharma Lane`, and line 7 displays the text `Asheville, NC 28899`.

Program 2-1

```java
1   public class ScreenOutput
2   {
3       public static void main(String[] args)
4       {
5           System.out.println("Kate Austen");
6           System.out.println("1234 Walnut Street");
7           System.out.println("Asheville, NC 28899");
8       }
9   }
```

Program Output
```
Kate Austen
1234 Walnut Street
Asheville, NC 28899
```

The statements that appear in lines 5 through 7 end with a semicolon. Just as a period marks the end of a sentence in English, a semicolon marks the end of a statement in Java. You'll notice that some lines of code in the program do not end with a semicolon, however. For example, class headers and method headers do not end with a semicolon because they are not considered statements. Also, the curly braces are not followed by a semicolon because they are not considered statements. (If this is confusing, don't despair! As you practice writing Java programs more and more, you will develop an intuitive understanding of the difference between statements and lines of code that are not considered statements.)

Notice that the output of the `System.out.println()` statements appear on separate lines. When the `System.out.println()` statement displays output, it advances the output cursor (the location where the next item of output will appear) to the next line. That means the `System.out.println()` statement displays its output, and then the next thing that is displayed will appear on the following line.

The `System.out.print()` statement displays output, but it does not advance the output cursor to the next line. Program 2-2 shows an example.

Program 2-2

```
1 public class ScreenOutput2
2 {
3    public static void main(String[] args)
4    {
5        System.out.print("Programming");
6        System.out.print("is");
7        System.out.print("fun.");
8    }
9 }
```

Program Output

```
Programmingisfun.
```

Oops! It appears from the program output that something went wrong. All of the words are jammed together into one long series of characters. If we want spaces to appear between the words, we have to explicitly display them. Program 2-3 shows how we have to insert spaces into the strings that we are displaying, if we want the words to be separated on the screen. Notice that in line 5 we have inserted a space in the string, after the letter g, and in line 6 we have inserted a space in the string after the letter s.

Program 2-3

```
1 public class ScreenOutput3
2 {
3    public static void main(String[] args)
4    {
5        System.out.print("Programming ");
6        System.out.print("is ");
7        System.out.print("fun.");
8    }
9 }
```

Program Output

```
Programming is fun.
```

Variables

In Java, variables must be declared before they can be used in a program. A variable declaration statement is written in the following general format:

DataType VariableName;

In the general format, *DataType* is the name of a Java data type, and *VariableName* is the name of the variable that you are declaring. The declaration statement ends with a semicolon. For example, the key word `int` is the name of the integer data type in Java, so the following statement declares a variable named `number`.

```
int number;
```

Table 2-1 lists the Java data types, gives their memory size in bytes, and describes the type of data that each can hold. Note that in this book we will primarily use the `int`, `double`, and `String` data types.[1]

Table 2-1 Java Data Types

Data Type	Size	What It Can Hold
`byte`	1 byte	Integers in the range of −128 to +127
`short`	2 bytes	Integers in the range of −32,768 to +32,767
`int`	4 bytes	Integers in the range of −2,147,483,648 to +2,147,483,647
`long`	8 bytes	Integers in the range of −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`float`	4 bytes	Floating-point numbers in the range of $\pm3.4\times10^{-38}$ to $\pm3.4\times10^{38}$, with 7 digits of accuracy
`double`	8 bytes	Floating-point numbers in the range of $\pm1.7\times10^{-308}$ to $\pm1.7\times10^{308}$, with 15 digits of accuracy
`String`	Varies	Strings of text.

Here are some other examples of variable declarations:

```
int speed;
double distance;
String name;
```

---

[1] Notice that `String` is written with an initial uppercase letter. To be correct, `String` is not a data type in Java, it is a class. We use it as a data type, though.

Several variables of the same data type can be declared with the same declaration statement. For example, the following statement declares three `int` variables named `width`, `height`, and `length`.

```
int width, height, length;
```

You can also initialize variables with starting values when you declare them. The following statement declares an `int` variable named `hours`, initialized with the starting value 40:

```
int hours = 40;
```

Variable Names

You may choose your own variable names (and class names) in Java, as long as you do not use any of the Java key words. The key words make up the core of the language and each has a specific purpose. Table 2-2 shows a complete list of Java key words.

The following are some specific rules that must be followed with all identifiers:

- The first character must be one of the letters a–z, A–Z, an underscore (_), or a dollar sign ($).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (_), or dollar signs ($).
- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Identifiers cannot include spaces.

Table 2-2 The Java Key Words

abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false	instanceof	protected	this	

Program 2-3 shows an example with three variable declarations. Line 5 declares a `String` variable named `name`, initialized with the string "Jeremy Smith". Line 6 declares an `int` variable named `hours` initialized with the value 40. Line 7 declares a `double` variable named `pay`, initialized with the value 852.99. Notice that in lines 9 through 11 we use `System.out.println` to display the contents of each variable.

Program 2-3
```
 1 public class VariableDemo
 2 {
 3    public static void main(String[] args)
 4    {
 5        String name = "Jeremy Smith";
 6        int hours = 40;
 7        double pay = 852.99;
 8
 9        System.out.println(name);
10        System.out.println(hours);
11        System.out.println(pay);
12    }
13 }
```
Program Output
```
Jeremy Smith
40
852.99
```

Reading Keyboard Input

To read keyboard input in Java you have to create a type of object in memory known as a `Scanner` object. You can then use the `Scanner` object to read values from the keyboard, and assign those values to variables. Program 2-4 shows an example of how this is done. (This is the Java version of pseudocode Program 2-2 in your textbook.) Let's take a closer look at the code:

- Line 1 has the following statement: `import java.util.Scanner;`
  This statement is necessary to tell the Java compiler that we are going to create a `Scanner` object in the program.
- Line 7 creates a `Scanner` object and gives it the name `keyboard`.
- Line 8 declares an `int` variable named `age`.
- Line 10 displays the string "`What is your age?`"
- Line 11 reads an integer value from the keyboard and assigns that value to the `age` variable.

- Line 12 displays the string "Here is the value that you entered:"
- Line 13 displays the value of the `age` variable.

Program 2-4

> This program is the Java version of **Program 2-2** in your textbook!

```java
 1 import java.util.Scanner;
 2
 3 public class GetAge
 4 {
 5    public static void main(String[] args)
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8       int age;
 9
10       System.out.println("What is your age?");
11       age = keyboard.nextInt();
12       System.out.println("Here is the value that you entered:");
13       System.out.println(age);
14    }
15 }
```

Program Output
```
What is your age?
24 [Enter]
Here is the value that you entered:
24
```

Notice that in line 11 we used the expression `keyboard.nextInt()` to read an integer from the keyboard. If we wanted to read a double from the keyboard, we would use the expression `keyboard.nextDouble()`. And, if we want to read a string from the keyboard, we would use the expression `keyboard.nextLine()`.

Program 2-5 shows how a `Scanner` object can be used to read not only integers, but `doubles` and strings:

- Line 7 creates a `Scanner` object and gives it the name `keyboard`.
- Line 8 declares a `String` variable named `name`, line 9 declares a `double` variable named `payRate`, and line 10 declares an `int` variable named `hours`.
- Line 13 uses the expression `keyboard.nextLine()` to read a string from the keyboard, and assigns the string to the `name` variable.
- Line 16 uses the expression `keyboard.nextDouble()` to read a `double` from the keyboard, and assigns it to the `payRate` variable.
- Line 19 uses the expression `keyboard.nextInt()` to read an integer from the keyboard, and assigns it to the `hours` variable.

Program 2-5

```java
 1 import java.util.Scanner;
 2
 3 public class GetInput
 4 {
 5     public static void main(String[] args)
 6     {
 7         Scanner keyboard = new Scanner(System.in);
 8         String name;
 9         double payRate;
10         int hours;
11
12         System.out.print("Enter your name: ");
13         name = keyboard.nextLine();
14
15         System.out.print("Enter your hourly pay rate: ");
16         payRate = keyboard.nextDouble();
17
18         System.out.print("Enter the number of hours worked: ");
19         hours = keyboard.nextInt();
20
21         System.out.println("Here are the values that you entered:");
22         System.out.println(name);
23         System.out.println(payRate);
24         System.out.println(hours);
25     }
26 }
```

Program Output

```
Enter your name: Connie Maroney [Enter]
Enter your hourly pay rate: 55.25 [Enter]
Enter the number of hours worked: 40 [Enter]
Here are the values that you entered:
Connie Maroney
55.25
40
```

Displaying Multiple Items with the + Operator

When the + operator is used with strings, it is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
System.out.println("This is " + "one string.");
```

This statement will display:

```
This is one string.
```

The + operator produces a string that is the combination of the two strings used as its operands. You can also use the + operator to concatenate the contents of a variable to a string. The following code shows an example:

```
number = 5;
System.out.println("The value is " + number);
```

The second line uses the + operator to concatenate the contents of the `number` variable with the string `"The value is "`. Although `number` is not a string, the + operator converts its value to a string and then concatenates that value with the first string. The output that will be displayed is:

```
The value is 5
```

Program 2-6 shows an example. (This is the Java version of the pseudocode Program 2-4 in your textbook.)

Program 2-6

```java
 1 import java.util.Scanner;
 2
 3 public class DisplayMultiple
 4 {
 5    public static void main(String[] args)
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8       int age;
 9       String name;
10
11       System.out.print("What is your name? ");
12       name = keyboard.nextLine();
13
14       System.out.print("What is your age? ");
15       age = keyboard.nextInt();
16
17       System.out.println("Hello " + name);
18       System.out.println("You are " + age + " years old.");
19    }
20 }
```

Program Output

```
What is your name? Tony [Enter]
What is your age? 49 [Enter]
Hello Tony
You are 49 years old.
```

Performing Calculations

Table 2-3 shows the Java arithmetic operators, which are nearly the same as those presented in Table 2-1 in your textbook.

Table 2-3 Java's Arithmetic Operators

Symbol	Operation	Description
+	Addition	Adds two numbers
−	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies two numbers
/	Division	Divides one number by another and gives the quotient
%	Modulus	Divides one integer by another and gives the remainder

Here are some examples of statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax;
sale = price - discount;
population = population * 2;
half = number / 2;
leftOver = 17 % 3;
```

Program 2-7 shows an example program that performs mathematical calculations (This program is the Java version of pseudocode Program 2-9 in your textbook.)

Perhaps you noticed that Table 2-3 does not show an exponent operator. Java does not provide such an operator, but it does provide a method named `Math.pow` for this purpose. Here is an example of how the `Math.pow` method is used:

```
result = Math.pow(4.0, 2.0);
```

The method takes two `double` arguments (the numbers shown inside the parentheses). It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

result = 4^2

Program 2-7

```java
 1 import java.util.Scanner;
 2
 3 public class CalcPercentage
 4 {
 5    public static void main(String[] args)
 6    {
 7       Scanner keyboard = new Scanner(System.in);
 8       double originalPrice, discount, salePrice;
 9
10       System.out.print("Enter the item's original price: ");
11       originalPrice = keyboard.nextDouble();
12
13       discount = originalPrice * 0.2;
14       salePrice = originalPrice - discount;
15
16       System.out.println("The sale price is $" + salePrice);
17    }
18 }
```

Program Output

```
Enter the item's original price: 100 [Enter]
The sale price is $80
```

Named Constants

You create named constants in Java by using the `final` key word in a variable declaration. The word `final` is written just before the data type. Here is an example:

```
final double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word `final` appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when declaring a variable with the `final` modifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a `final` variable.

Documenting a Program with Comments

To write a line comment in Java you simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Here is an example:

```
// This program calculates an employee's gross pay.
```

Multi-line comments, or block comments, start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Here is an example:

```
/*
    This program calculates an employee's gross pay.
    Written by Matt Hoyle.
*/
```

Chapter 3   Methods

Chapter 3 in your textbook discusses modules as named groups of statements that perform specific tasks in a program. In Java, modules are called *methods*. In this chapter we will discuss how to define and call Java methods, declare local variables in a method, and pass arguments to a method. We will also discuss the declaration of class fields, and how they can be used to create constants that are visible to all of the methods in a class.

Defining and Calling Methods

To create a method you must write its *definition*, which consists of two general parts: a header and a body. As you learned in Chapter 2, the *method header* is the line that appears at the beginning of a method definition. It lists several things about the method, including the method's name. The *method body* is a collection of statements that are performed when the method is executed. These statements are enclosed inside a set of curly braces.

As you already know, every complete Java program must have a `main` method. Java programs can have other methods as well. Here is an example of a simple method that displays a message on the screen:

```
public static void showMessage()
{
    System.out.println("Hello world");
}
```

For now, the headers for all of the Java methods that you will write will begin with the key words `public static void`. Following this you write the name of the method, and a set of parentheses. Remember that a method header never ends with a semicolon!

Calling a Method

A method executes when it is called. The `main` method is automatically called when a program starts, but other methods are executed by method call statements. When a method is called, the program branches to that method and executes the statements in its body. Here is an example of a method call statement that calls the `showMessage` method we previously examined:

```
showMessage();
```

The statement is simply the name of the method followed by a set of parentheses. Because it is a complete statement, it is terminated with a semicolon.

Program 3-1 shows a Java program that demonstrates the `showMessage` method. This is the Java version of pseudocode Program 3-1 in your textbook.

Program 3-1

> This program is the Java version of **Program 3-1** in your textbook!

```java
 1 public class SimpleMethodDemo
 2 {
 3    public static void main(String[] args)
 4    {
 5       System.out.println("I have a message for you.");
 6       showMessage();
 7       System.out.println("That's all, folks!");
 8    }
 9
10    public static void showMessage()
11    {
12       System.out.println("Hello world");
13    }
14 }
```

Program Output
```
I have a message for you.
Hello world
That's all, folks!
```

The class that contains this program's code has two methods: `main` and `showMessage`. The `main` method appears in lines 3 through 8, and the `showMessage` method appears in lines 10 through 13. When the program runs, the `main` method executes. The statement in line 5 displays "I have a message for you." Then the statement in line 6 calls the `showMessage` method. This causes the program to branch to the `showMessage` method and execute the statement that appears in line 12. This displays "Hello world". The program then branches back to the `main` method and resumes execution at line 7. This displays "That's all, folks!"

Local Variables

Variables that are declared inside a method are known as local variables. They are called *local* because they are local to the method in which they are declared. Statements outside a method cannot access that method's local variables.

Because a method's local variables are hidden from other methods, the other methods may have their own local variables with the same name. For example, look at Program 3-2. In addition to the `main` method, this program has two other methods: `texas` and `california`. These two methods each have a local variable named `birds`.

Program 3-2

```java
 1 public class LocalVarDemo
 2 {
 3    public static void main(String[] args)
 4    {
 5       // Call the texas method.
 6       texas();
 7
 8       // Call the california method.
 9       california();
10    }
11
12    // Definition of the texas method
13    public static void texas()
14    {
15       // Local variable named birds
16       int birds = 1000;
17
18       // Display the value of the birds variable.
19       System.out.println("The texas method has " +
20                          birds + " birds.");
21    }
22
23    // Definition of the california method
24    public static void california()
25    {
26       // Local variable named birds
27       int birds = 200;
28
29       // Display the value of the birds variable.
30       System.out.println("The california method has " +
31                          birds + " birds.");
32    }
33 }
```

Program Output

```
The texas method has 1000 birds.
The california method has 200 birds.
```

Although there are two variables named `birds`, the program can only see one of them at a time because they are in different methods. When the `texas` method is executing, the `birds` variable declared inside `texas` is visible. When the `california` method is executing, the `birds` variable declared inside `california` is visible.

It's worth noting that although different methods can have a local variable with the same name, you cannot declare two local variables with the same name in the same method.

Passing Arguments to Methods

If you want to be able to pass an argument into a method, you must declare a parameter variable in that method's header. The parameter variable will receive the argument that is passed when the method is called. Here is the definition of a method that uses a parameter:

```
public static void displayValue(int num)
{
    System.out.println("The value is " + num);
}
```

Notice the integer variable declaration that appears inside the parentheses (`int num`). This is the declaration of a parameter variable, which enables the `displayValue` method to accept an integer value as an argument. Here is an example of a call to the `displayValue` method, passing 5 as an argument:

```
displayValue(5);
```

This statement executes the `displayValue` method. The argument that is listed inside the parentheses is copied into the method's parameter variable, `num`.

Program 3-3 shows a complete program with a method that accepts an argument. This is the Java version of pseudocode Program 3-5 in your textbook. When the program runs, it prompts the user to enter a number. Line 18 reads an integer from the keyboard and assigns it to the `number` variable. Line 22 calls the `doubleNumber` method, passing the `number` variable as an argument.

The `doubleNumber` method is defined in lines 26 through 36. The method has an `int` parameter variable named `value`. A local `int` variable named `result` is declared in line 29, and in line 32 the `value` parameter is multiplied by 2 and the result is assigned to the `result` variable. In line 35 the value of the `result` variable is displayed.

Program 3-3

```java
1 import java.util.Scanner;
2
3 public class PassArgument
4 {
5    public static void main(String[] args)
6    {
7       // Create a Scanner object for keyboard input.
8       Scanner keyboard = new Scanner(System.in);
9
10      // Declare a variable to hold a number.
11      int number;
12
13      // Prompt the user for a number
14      System.out.println("Enter a number and I will display");
15      System.out.println("that number doubled.");
16
17      // Read an integer from the keyboard.
18      number = keyboard.nextInt();
19
20      // Call the doubleNumber method passing
21      // number as an argument.
22      doubleNumber(number);
23   }
24
25   // Definition of the doubleNumber method
26   public static void doubleNumber(int value)
27   {
28      // Local variable to hold the result
29      int result;
30
31      // Multiply the value parameter times 2.
32      result = value * 2;
33
34      // Display the result.
35      System.out.println(result);
36   }
37 }
```

Program Output

```
Enter a number and I will display
that number doubled.
```
22 [*Enter*]
```
44
```

Passing Multiple Arguments

Often it is useful to pass more than one argument to a method. When you define a method, you must declare a parameter variable for each argument that you want passed into the method. Program 3-4 shows an example. This is the Java version of pseudocode Program 3-6 in your textbook.

Program 3-4

This is the Java version of **Program 3-6** in your textbook!

```java
 1 public class MultipleArgs
 2 {
 3    public static void main(String[] args)
 4    {
 5        System.out.println("The sum of 12 and 45 is:");
 6        showSum(12, 45);
 7    }
 8
 9    public static void showSum(int num1, int num2)
10    {
11        int result;
12        result = num1 + num2;
13        System.out.println(result);
14    }
15 }
```

Program Output

```
The sum of 12 and 45 is:
57
```

Arguments Are Passed by Value

In Java, all arguments of the built-in data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable. A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If a parameter variable is changed inside a method, it has no effect on the original argument.

Global Constants

In Java, variables and constants cannot be declared outside of a class. If you declare a variable or constant inside a class, but outside of all the class's methods, that variable or constant is known as a *class field*, and will be available to all of the methods within the class. Program 3-5 demonstrates how to declare such a constant. Notice that in line 3 we have declared a constant named INTEREST_RATE .

The declaration is inside the class, but it is not inside any of the methods. As a result, the constant is available to all of the methods in the class. Also notice that the declaration begins with the key words `public static`. At this point you do not need to be too concerned about why these key words are required, except that this makes the constant available to all `public static` methods.

Program 3-5

```
 1 public class AccountProcessor
 2 {
 3    public static final double INTEREST_RATE = 0.05;
 4
 5    public static void main(String[] args)
 6    {
 7       // Statements here have access to
 8       // the INTEREST_RATE constant.
 9    }
10
11    public static void method2()
12    {
13       // Statements here have access to
14       // the INTEREST_RATE constant.
15    }
17    public static void method3()
18    {
19       // Statements here have access to
20       // the INTEREST_RATE constant.
21    }
22 }
```

Chapter 4   Decision Structures and Boolean Logic

Relational Operators and the `if` Statement

Java's relational operators, shown in Table 4-1, are exactly the same as those discussed in your textbook.

Table 4-1 Relational Operators

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

The relational operators are used to create Boolean expressions, and are commonly used with if statements. Here is the general format of the `if` statement in Java:

```
if (BooleanExpression)
{
   statement;
   statement;
   etc;
}
```

The statement begins with the word `if`, followed by a Boolean expression that is enclosed in a set of parentheses. Beginning on the next line is a set of statements that are enclosed in curly braces. When the `if` statement executes, it evaluates the Boolean expression. If the expression is true, the statements inside the curly braces are executed. If the Boolean expression is false, the statements inside the curly braces are skipped. We sometimes say that the statements inside the curly braces are conditionally executed because they are executed only under the condition that the Boolean expression is true.

If you are writing an `if` statement that has only one conditionally executed statement, you do not have to enclose the conditionally executed statement inside curly braces. Such an `if` statement can be written in the following general format:

```
if (BooleanExpression)
   statement;
```

When an `if` statement written in this format executes, the Boolean expression is tested. If it is true, the one statement that appears on the next line will be executed. If the Boolean expression is false, however, that one statement is skipped.

Program 4-1 demonstrates the `if` statement. This is the Java version of pseudocode Program 4-1 in your textbook.

Program 4-1

```java
 1 import java.util.Scanner;
 2
 3 public class AverageScore
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Create a Scanner object for keyboard input.
 8       Scanner keyboard = new Scanner(System.in);
 9
10       // Declare variables
11       double test1, test2, test3, average;
12
13       // Get test 1
14       System.out.print("Enter the score for test #1: ");
15       test1 = keyboard.nextDouble();
16
17       // Get test 2
18       System.out.print("Enter the score for test #2: ");
19       test2 = keyboard.nextDouble();
20
21       // Get test 3
22       System.out.print("Enter the score for test #3: ");
23       test3 = keyboard.nextDouble();
24
25       // Calculate the average score.
26       average = (test1 + test2 + test3) / 3;
27
28       // Display the average.
29       System.out.println("The average is " + average);
30
31       // If the average is greater than 95
32       // congratulate the user.
33       if (average > 95)
34          System.out.println("Way to go! Great average!");
35    }
36 }
```

This is the Java version of **Program 4-1** in your textbook!

Program Output

```
Enter the score for test #1: 100 [Enter]
Enter the score for test #2: 90 [Enter]
```

```
Enter the score for test #3: 95 [Enter]
The average is 95.0
```

Dual Alternative Decision Structures

You use the `if-else` statement in Java to create a dual alternative decision structure. This is the format of the `if-else` statement:

```
if (BooleanExpression)
{
    statement;
    statement;
    etc;
}
else
{
    statement;
    statement;
    etc;
}
```

An `if-else` statement has two parts: an `if` clause and an `else` clause. Just like a regular `if` statement, the `if-else` statement tests a Boolean expression. If the expression is true, the block of statements following the `if` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement. If the Boolean expression is false, the block of statements following the `else` clause is executed, and then control of the program jumps to the statement that follows the `if-else` statement.

The `if-else` statement has two sets of conditionally executed statements. One set is executed only under the condition that the Boolean expression is true, and the other set is executed only under the condition that the Boolean expression is false. Under no circumstances will both sets of conditionally executed statement be executed.

If either set of conditionally executed statements contains only one statement, the curly braces are not required. For example, the following general format shows only one statement following the `if` clause and only one statement following the `else` clause:

```
if (BooleanExpression)
    statement;
else
    statement;
```

Program 4-2 shows an example. This is the Java version of pseudocode Program 4-2 in your textbook. The program gets the number of hours that the user has worked (line 19) and the user's hourly pay rate (line 23). The `if-else` statement in lines 26 through 29 determines whether the user has worked more than 40 hours. If so, the `calcPayWithOT` method is called in line 27. Otherwise the `calcRegularPay` method is called in line 29.

Program 4-2

```java
1  import java.util.Scanner;
2
3  public class DualAlternative
4  {
5      // Globally visible constants.
6      public final static int BASE_HOURS = 40;
7      public final static double OT_MULTIPLIER = 1.5;
8
9      public static void main(String[] args)
10     {
11         // Create a Scanner object for keyboard input.
12         Scanner keyboard = new Scanner(System.in);
13
14         // Declare local variables
15         double hoursWorked, payRate, grossPay;
16
17         // Get the number of hours worked.
18         System.out.print("Enter the number of hours worked: ");
19         hoursWorked = keyboard.nextInt();
20
21         // Get the hourly pay rate.
22         System.out.print("Enter the hourly pay rate: ");
23         payRate = keyboard.nextDouble();
24
25         // Calculate the gross pay.
26         if (hoursWorked > BASE_HOURS)
27             grossPay = calcPayWithOT(hoursWorked, payRate);
28         else
29             grossPay = calcRegularPay(hoursWorked, payRate);
30
31         // Display the gross pay.
32         System.out.println("The gross pay is $" + grossPay);
33     }
34
35     // The calcPayWithOT function calculates pay
36     // with overtime and returns that value.
37     public static double calcPayWithOT(double hours, double rate)
```

This is the Java version of **Program 4-2** in your textbook!

```
38   {
39       // Local variables
40       double overtimeHours, overtimePay, gross;
41
42       // Calculate the number of overtime hours.
43       overtimeHours = hours - BASE_HOURS;
44
45       // Calculate the overtime pay
46       overtimePay = overtimeHours * rate * OT_MULTIPLIER;
47
48       // Calculate the gross pay.
49       gross = BASE_HOURS * rate + overtimePay;
50
51       return gross;
52   }
53
54   // The calcRegularPay module calculates
55   // pay with no overtime and returns that value.
56   public static double calcRegularPay(double hours, double rate)
57   {
58       return hours * rate;
59   }
60 }
```

Program Output

```
Enter the number of hours worked: 100 [Enter]
Enter the hourly pay rate: 10 [Enter]
The gross pay is $1300.0
```

Comparing Strings

In Java you do not use the relational operators to compare strings. To determine whether `string1` is equal to `string2`, you use the following notation:

> *string1*.equals(*string2*)

This is an expression that returns true if `string1` is equal to `string2`. Otherwise, it returns false. Program 4-3 shows an example. This is the Java version of pseudocode Program 4-3 in your textbook.

The `if-else` statement that begins in line 19 tests the expression `password.equals("prospero")`. This expression will return true if the contents of the password variable is equal to "prospero", or false otherwise.

Program 4-3

```java
 1 import java.util.Scanner;
 2
 3 public class StringEqual
 4 {
 5    public static void main(String[] args)
 6    {
 7        // Create a Scanner object for keyboard input.
 8        Scanner keyboard = new Scanner(System.in);
 9
10        // A variable to hold a password.
11        String password;
12
13        // Prompt the user to enter the password.
14        System.out.print("Enter the password: ");
15        password = keyboard.nextLine();
16
17        // Determine whether the correct password
18        // was entered.
19        if (password.equals("prospero"))
20            System.out.print("Password accepted.");
21        else
22            System.out.print(""Sorry, that is not the correct password.");
23    }
24 }
```

This is the Java version of **Program 4-3** in your textbook!

Program Output

Enter the password: **ferdinand [*Enter*]**
Sorry, that is not the correct password.

Program Output

Enter the password: **prospero [*Enter*]**
Password accepted.

You can also determine whether *string1* is less-than, equal-to, or greater-than *string2* using the following notation:

> *string1*.compareTo(*string2*)

This expression returns an integer value that can be used in the following manner:

- If the expression's value is negative, then *string1* is less than *string2*.
- If the expression's value is 0, then *string1* and *string2* are equal.
- If the expression's value is is positive, then *string1* is greater than *string2*.

For example, assume that `name1` and `name2` are `String` variables. The following `if` statement compares the strings:

```
if (name1.compareTo(name2) == 0)
   System.out.println("The names are the same.");
```

Also, the following statement compares `name1` to the string literal "Joe":

```
if (name1.compareTo("Joe") == 0)
   System.out.println("The names are the same.");
```

Nested Decision Structures

Program 4-5 shows an example of nested decision structures. As noted in your textbook, this type of nested decision structure can also be written as an `if-else-if` statement, as shown in Program 4-6.

Program 4-5
```
 1 import java.util.Scanner;
 2
 3 public class NestedIf
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Create a Scanner object for keyboard input.
 8       Scanner keyboard = new Scanner(System.in);
 9
10       // A variable to hold the temperature.
11       int temp;
12
13       // Prompt the user to enter the temperature.
14       System.out.print("What is the outside temperature? ");
15       temp = keyboard.nextInt();
16
17       // Determine the type of weather we're having.
18       if (temp < 30)
19          System.out.println("Wow! That's cold!");
20       else
21       {
22          if (temp < 50)
23             System.out.println("A little chilly.");
24          else
25          {
26             if (temp < 80)
```

```
27                    System.out.println("Nice and warm.");
28               else
29                    System.out.println("Whew! It's hot!");
30            }
31        }
32     }
33 }
```

Program Output

```
What is the outside temperature? 20 [Enter]
Wow! That's cold!
```

Program Output

```
What is the outside temperature? 45 [Enter]
A little chilly.
```

Program Output

```
What is the outside temperature? 70 [Enter]
Nice and warm.
```

Program Output

```
What is the outside temperature? 90 [Enter]
Whew! Its hot!
```

Program 4-6

```java
 1 import java.util.Scanner;
 2
 3 public class IfElseIf
 4 {
 5    public static void main(String[] args)
 6    {
 7        // Create a Scanner object for keyboard input.
 8        Scanner keyboard = new Scanner(System.in);
 9
10        // A variable to hold the temperature.
11        int temp;
12
13        // Prompt the user to enter the temperature.
14        System.out.print("What is the outside temperature? ");
15        temp = keyboard.nextInt();
16
17        // Determine the type of weather we're having.
```

```
18        if (temp < 30)
19            System.out.println("Wow! That's cold!");
20        else if (temp < 50)
21                System.out.println("A little chilly.");
22        else if (temp < 80)
23                System.out.println("Nice and warm.");
24        else
25                System.out.println("Whew! it's hot!");
26    }
27 }
```

Program Output

```
What is the outside temperature? 20 [Enter]
Wow! That's cold!
```

Program Output

```
What is the outside temperature? 45 [Enter]
A little chilly.
```

Program Output

```
What is the outside temperature? 70 [Enter]
Nice and warm.
```

Program Output

```
What is the outside temperature? 90 [Enter]
Whew! Its hot!
```

The Case Structure (`switch` Statement)

In Java, case structures are written as `switch` statements. Here is the general format of the `switch` statement:

```
switch (testExpression)          ◄────── This is a variable or an expression.
{
    case value_1:
        statement
        statement            These statements are executed if the
        etc.                 testExpression is equal to value_1.
        break;

    case value_2:
        statement
        statement            These statements are executed if the
        etc.                 testExpression is equal to value_2.
        break;

    Insert as many case sections as necessary

    case value_N:
        statement
        statement            These statements are executed if the
        etc.                 testExpression is equal to value_N.
        break;

    default:
        statement            These statements are executed if the
        statement            testExpression is not equal to any of the values
        etc.                 listed after the Case statements.
}                    ◄──────      This is the end of the switch statement.
```

The first line of the structure starts with the word `switch`, followed by a *testExpression* which is encloses in parentheses. The *testExpression* is a value or expression of one of these types: `char`, `byte`, `short`, or `int`. (Beginning with Java 7, it can also be a string.) Beginning at the next line is a block of code enclosed in curly braces. Inside this block of code is one or more *case sections*. A `case` section begins with the word `case`, followed by a value, followed by a colon. Each `case` section contains one or more statements, followed by a `break` statement. At the end is an optional *default section*.

When the `switch` statement executes, it compares the value of the *testExpression* with the values that follow each of the `case` statements (from top to bottom). When it finds a `case` value that matches the *testExpression*'s value, the program branches to the `case` statement. The statements that follow the `case` statement are executed, until a break statement is encountered. At that point program jumps out of the `switch` statement. If the *testExpression* does not match any of the `case` values, the program branches to the `default` statement and executes the statements that immediately following it.

For example, the following code performs the same operation as the flowchart in Figure 4-18:

```
switch (month)
{
    case 1:
        System.out.println("January");
        break;

    case 2:
        System.out.println("February");
        break;

    case 3:
        System.out.println("March");
        break;

    default:
        System.out.println("Error: Invalid month");
}
```

In this example the *testExpression* is the `month` variable. If the value in the `month` variable is 1, the program will branch to the `case 1:` section and execute the `System.out.println("January")` statement that immediately follows it. If the value in the `month` variable is 2, the program will branch to the `case 2:` section and execute the `System.out.println("February")` statement that immediately follows it. If the value in the `month` variable is 3, the program will branch to the `case 3:` section and execute the `System.out.println("March")` statement that immediately follows it. If the value in the `month` variable is not 1, 2, or 3, the program will branch to the `default:` section and execute the `System.out.println("Error: Invalid month")` statement that immediately follows it.

Here are some important points to remember about the `switch` statement:

- The *testExpression* must be a value or expression of one of these types: `char`, `byte`, `short`, or `int`. If you are using Java 7, the *testExpression* can also be a string.

- The value that follows a `case` statement must be a literal or a named constant of one of these types: `char`, `byte`, `short`, or `int`. Beginning with Java 7, it can also be a string.
- The `break` statement that appears at the end of a `case` section is optional, but in most situations you will need it. If the program executes a `case` section that does not end with a `break` statement, it will continue executing the code in the very next `case` section.
- The `default` section is optional, but in most situations you should have one. The `default` section is executed when the *testExpression* does not match any of the `case` values.
- Because the `default` section appears at the end of the switch statement, it does not need a `break` statement.

Program 4-7 shows a complete example. This is the Java version of pseudocode Program 4-8 in your textbook.

Program 4-7

```
 1 import java.util.Scanner;
 2
 3 public class TVModels
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Create a Scanner object for keyboard input.
 8       Scanner keyboard = new Scanner(System.in);
 9
10       // Constants for the TV prices
11       final double MODEL_100_PRICE = 199.99;
12       final double MODEL_200_PRICE = 269.99;
13       final double MODEL_300_PRICE = 349.99;
14
15       // Constants for the TV sizes
16       final int MODEL_100_SIZE = 24;
17       final int MODEL_200_SIZE = 27;
18       final int MODEL_300_SIZE = 32;
19
20       // Variable for the model number
21       int modelNumber;
22
23       // Get the model number.
24       System.out.print("Which TV are you interested in? ");
25       System.out.print("The 100, 200, or 300? ");
26       modelNumber = keyboard.nextInt();
27
```

This is the Java version of Program 4-8 in your textbook!

```
28          // Display the price and size.
29          switch (modelNumber)
30          {
31              case 100:
32                  System.out.println("Price: $" + MODEL_100_PRICE);
33                  System.out.println("Size: " + MODEL_100_SIZE);
34                  break;
35              case 200:
36                  System.out.println("Price: $" + MODEL_200_PRICE);
37                  System.out.println("Size: " + MODEL_200_SIZE);
38                  break;
39              case 300:
40                  System.out.println("Price $" + MODEL_300_PRICE);
41                  System.out.println("Size: " + MODEL_300_SIZE);
42                  break;
43              default:
44                  System.out.println("Invalid model number.");
45          }
46      }
47  }
```

Program Output

```
Which TV are you interested in? The 100, 200, or 300? 100 [Enter]
Price: $199.99
Size: 24
```

Program Output

```
Which TV are you interested in? The 100, 200, or 300? 200 [Enter]
Price: $269.99
Size: 27
```

Program Output

```
Which TV are you interested in? The 100, 200, or 300? 300 [Enter]
Price: $349.99
Size: 32
```

Program Output

```
Which TV are you interested in? The 100, 200, or 300? 500 [Enter]
Invalid model number.
```

Logical Operators

Java's logical operators look different than the ones used in your textbook's pseudocode, but they work in the same manner. Table 4-2 shows Java's logical operators.

Table 4-2 Java's Logical Operators

Operator	Meaning
&&	This is the logical AND operator. It connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
||	This is the logical OR operator. It connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
!	This is the logical NOT operator. It is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The ! operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

For example, the following `if` statement checks the value in `x` to determine if it is in the range of 20 through 40:

```
if (x >= 20 && x <= 40)
    System.out.println(x + " is in the acceptable range.");
```

The Boolean expression in the `if` statement will be true only when `x` is greater than or equal to 20 AND less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be true. The following statement determines whether `x` is outside the range of 20 through 40:

```
if (x < 20 || x > 40)
    System.out.println(x + " is outside the acceptable range.");
```

Here is an `if` statement using the `!` operator:

```
if (!(temperature > 100))
    System.out.println("This is below the maximum temperature.");
```

First, the expression `(temperature > 100)` is tested and a value of either true or false is the result. Then the `!` operator is applied to that value. If the expression `(temperature > 100)` is true, the `!` operator returns false. If the expression `(temperature > 100)` is false, the `!` operator returns true. The previous code is equivalent to asking: "Is the temperature not greater than 100?"

Boolean Variables

In Java you use the `boolean` data type to create Boolean variables. A `boolean` variable can hold one of two possible values: `true` or `false`. Here is an example of a `boolean` variable declaration:

```
boolean highScore;
```
Boolean variables are commonly used as flags that signal when some condition exists in the program. When the flag variable is set to false, it indicates the condition does not yet exist. When the flag variable is set to true, it means the condition does exist.

For example, suppose a test grading program has a `boolean` variable named `highScore`. The variable might be used to signal that a high score has been achieved by the following code:

```
if (average > 95)
   highScore = true;
```

Later, the same program might use code similar to the following to test the `highScore` variable, in order to determine whether a high score has been achieved:

```
if (highScore)
   System.out.println("That's a high score!");
```

Chapter 5   Repetition Structures

Incrementing and Decrementing Variables

To *increment* a variable means to increase its value and to *decrement* a variable means to decrease its value. Both of the following statements increment the variable `num` by one:

```
num = num + 1;
num += 1;
```

And `num` is decremented by one in both of the following statements:

```
num = num - 1;
num -= 1;
```

Incrementing and decrementing is so commonly done in programs that Java provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is `++` and the decrement operator is `--`. The following statement uses the `++` operator to add 1 to `num`:

```
num++;
```

After this statement executes, the value of `num` will increased by one. The following statement uses the `--` operator to subtract 1 from `num`:

```
num--;
```

In these examples, we have written the `++` and `--` operators after their operands (or, on the right side of their operands). This is called *postfix mode*. The operators can also be written before (or, on the left side) of their operands, which is called *prefix mode*. Here are examples:

```
++num;
--num;
```

When you write a simple statement to increment or decrement a variable, such as the ones shown here, it doesn't matter if you use prefix mode or postfix mode. The operators do the same thing in either mode. However, if you write statements that mix these operators with other operators or with other operations, there is a difference in the way the two modes work. Such complex code can be difficult to understand and debug. When we use the increment and decrement operators, we will do so only in ways that are straightforward and easy to understand, such as the statements previously shown.

We introduce these operators at this point because they are commonly used in certain types of loops. When we discuss the `for` loop you will see these operators used often.

The `while` Loop

In Java, the `while` loop is written in the following general format:

```
while (BooleanExpression)
{
    statement;
    statement;
    etc;
}
```

We will refer to the first line as the *while clause*. The `while` clause begins with the word `while`, followed by a Boolean expression that is enclosed in parentheses. Beginning on the next line is a block of statements that are enclosed in curly braces. This block of statements is known as the *body* of the loop.

When the `while` loop executes, the Boolean expression is tested. If the Boolean expression is true, the statements that appear in the body of the loop are executed, and then the loop starts over. If the Boolean expression is false, the loop ends and the program resumes execution at the statement immediately following the loop.

We say that the statements in the body of the loop are conditionally executed because they are executed only under the condition that the Boolean expression is true. If you are writing a `while` loop that has only one statement in its body, you do not have to enclose the statement inside curly braces. Such a loop can be written in the following general format:

```
while (BooleanExpression)
    statement;
```

When a `while` loop written in this format executes, the Boolean expression is tested. If it is true, the one statement that appears on the next line will be executed, and then the loop starts over. If the Boolean expression is false, however, the loop ends.

Program 5-1 shows an example of the `while` loop. This is the Java version of pseudocode Program 5-2 in your textbook.

Program 5-1

```java
 1 import java.util.Scanner;
 2
 3 public class ChemicalLabs
 4 {
 5     public static void main(String[] args)
 6     {
 7         // Create a Scanner object for keyboard input.
 8         Scanner keyboard = new Scanner(System.in);
 9
10         // Variable to hold the temperature
11         double temperature;
12
13         // Constant for the maximum temperature
14         final double MAX_TEMP = 102.5;
15
16         // Get the substance's temperature.
17         System.out.print("Enter the substance's temperature: ");
18         temperature = keyboard.nextDouble();
19
20         // If necessary, adjust the thermostat.
21         while (temperature > MAX_TEMP)
22         {
23             System.out.println("The temperature is too high.");
24             System.out.println("Turn the thermostat down and wait");
25             System.out.println("five minutes. Take the temperature");
26             System.out.print("again and enter it here: ");
27             temperature = keyboard.nextDouble();
28         }
29
30         // Remind the user to check the temperature
31         // again in 15 minutes.
32         System.out.println("The temperature is acceptable.");
33         System.out.println("Check it again in 15 minutes.");
34     }
35 }
```

Program Output

```
Enter the substance's temperature: 200 [Enter]
The temperature is too high.
Turn the thermostat down and wait
five minutes. Take the temperature
again and enter it here: 130 [Enter]
The temperature is too high.
```

```
Turn the thermostat down and wait
five minutes. Take the temperature
again and enter it here: 100 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

The `do-while` Loop

Here is the general format of the do-while loop:

```
do
{
    statement;
    statement;
    etc;
} while (BooleanExpression);
```

As with the while loop, the braces are optional if there is only one statement in the body of the loop. This is the general format of the do-while loop with only one conditionally executed statement:

```
do
    statement;
while (BooleanExpression);
```

Notice that a semicolon appears at the very end of the do-while statement. This semicolon is required, and leaving it out is a common error.

The `for` Loop

The for loop is specifically designed to initialize, test, and increment a counter variable. Here is the general format of the for loop:

```
for (InitializationExpression; TestExpression; IncrementExpression)
{
    statement;
    statement;
    etc.
}
```

The statements that appear inside the curly braces are the body of the loop. These are the statements that are executed each time the loop iterates. As with other control structures, the curly braces are optional if the body of the loop contains only one statement, as shown in the following general format:

```
for (InitializationExpression; TestExpression; IncrementExpression)
    statement;
```

The first line of the `for` loop is the *loop header*. After the key word `for`, there are three expressions inside the parentheses, separated by semicolons. (Notice there is not a semicolon after the third expression.)

The first expression is the *initialization expression*. It is normally used to initialize a counter variable to its starting value. This is the first action performed by the loop, and it is only done once. The second expression is the *test expression*. This is a Boolean expression that controls the execution of the loop. As long as this expression is true, the body of the `for` loop will repeat. The `for` loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the *increment expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's counter variable.

Here is an example of a simple `for` loop that prints "Hello" five times:

```
for (count = 1; count <= 5; count++)
{
    System.out.println("Hello");
}
```

In this loop, the initialization expression is count = 1, the test expression is count <= 5, and the increment expression is count++. The body of the loop has one statement, which is the call to dbPrint. This is a summary of what happens when this loop executes:

(1)   The initialization expression `count = 1` is executed. This assigns 1 to the `count` variable.
(2)   The expression `count <= 5` is tested. If the expression is true, continue with step 3. Otherwise, the loop is finished.
(3)   The statement `System.out.println("Hello");` is executed.
(4)   The increment expression `count++` is executed. This adds 1 to the `count` variable.
(5)   Go back to step 2.

Program 5-2 shows an example. This is the Java version of pseudocode Program 5-8 in your textbook.

Program 5-2

```java
1 public class ForLoop
2 {
3     public static void main(String[] args)
4     {
5         int counter;
6         final int MAX_VALUE = 5;
7
8         for (counter = 1; counter <= MAX_VALUE; counter++)
9         {
10            System.out.println("Hello world");
11        }
12    }
13 }
```

This is the Java version of **Program 5-8** in your textbook!

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

Program 5-3 shows another example. The `for` loop in this program uses the value of the counter variable in a calculation in the body of the loop. This is the Java version of pseudocode Program 5-9 in your textbook.

I should point out the "`\t`" formatting characters that are used in lines 12 and 20. These are special formatting characters known as the *tab escape sequence*. The escape sequence works similarly to the word `Tab` that is used in pseudocode in your textbook. As you can see in the program output, the "`\t`" characters are not displayed on the screen, but rather cause the output cursor to "tab over." It is useful for aligning output in columns on the screen.

Program 5-3

```java
1 public class ForLoop2
2 {
3     public static void main(String[] args)
4     {
5         // Variables
6         int counter, square;
7
8         // Constant for the maximum value
9         final int MAX_VALUE = 10;
10
11        // Display table headings.
12        System.out.println("Number\tSquare");
13        System.out.println("----------------------");
14
15        // Display the numbers 1 through 10 and
```

This is the Java version of **Program 5-9** in your textbook!

Page 47

```
16         // their squares.
17         for (counter = 1; counter <= MAX_VALUE; counter++)
18         {
19             square = counter * counter;
20             System.out.println(counter + "\t\t" + square);
21         }
22     }
23 }
```

Program Output
```
Number          Square
-----------------------
1               1
2               4
3               9
4               16
5               25
6               36
7               49
8               64
9               81
10              100
```

Incrementing by Values Other Than 1

Program 5-4 demonstrates that the update expression does not have to increment the counter variable by 1. In fact, the update expression can be any expression that you wish to execute at the end of each loop iteration. In Program 5-4 , the counter variable is incremented by 2, causing the statement in line 15 to display only the odd numbers in the range of 1 through 11. This program is the Java version of pseudocode Program 5-10 in your textbook.

Program 5-4
```
 1 import java.util.Scanner;
 2
 3 public class ForLoop3
 4 {
 5     public static void main(String[] args)
 6     {
 7         // Declare a counter variable.
 8         int counter;
 9
10         // Constant for the maximum value
11         final int MAX_VALUE = 11;
12
13         for (counter = 1; counter <= MAX_VALUE; counter = counter + 2)
14         {
15             System.out.println(counter);
16         }
17     }
18 }
```

This is the Java version of **Program 5-10** in your textbook!

```
Program Output
1
3
5
7
9
11
```

Calculating a Running Total

Your textbook discusses the common programming task of calculating the sum of a series of values, also
known as calculating a running total. Program 5-5 demonstrates how this is done in Java. The `total`
variable that is declared in line 16 is the accumulator variable. Notice that it is initialized with the value
0. During each loop iteration the user enters a number, and in line 30 this number is added to the value
already stored in `total`. The `total` variable accumulates the sum of all the numbers entered by the
user. This program is the Java version of pseudocode Program 5-18 in your textbook.

Program 5-5
```java
 1 import java.util.Scanner;
 2
 3 public class RunningTotal
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Create a Scanner object for keyboard input.
 8       Scanner keyboard = new Scanner(System.in);
 9
10       // Declare a variable to hold each number
11       // entered by the user.
12       int number;
13
14       // Declare an accumulator variable,
15       // initialized with 0.
16       int total = 0;
17
18       // Declare a counter variable for the loop.
19       int counter;
20
21       // Explain what we are doing.
22       System.out.println("This program calculates the");
23       System.out.println("total of five numbers.");
24
25       // Get five numbers and accumulate them.
26       for (counter = 1; counter <= 5; counter++)
27       {
28          System.out.print("Enter a number: ");
29          number = keyboard.nextInt();
30          total = total + number;
31       }
32
33       // Display the total of the numbers.
34       System.out.println("The total is " + total);
```

This is the Java version of
Program 5-18 in your textbook!

```
35    }
36 }
```
Program Output
```
1
3
5
7
9
11
```

Chapter 6   Value-Returning Methods

Terminology

Chapter 6 in your textbook is about functions, which are modules that return a value. In Java, we use the term method for both modules and functions. A module in Java is simply a method that does not return a value (also known as a void method), and a function is a value-returning method.

Generating Random Integers

To generate random numbers in Java you create a type of object in memory known as a `Random` object. You can then use the `Random` object to generate random numbers. First, you write the following statement at the top of your program:

```
import java.util.Random;
```

Then, at the point where you wish to create the `Random` object you write a statement like this:

```
Random randomNumbers = new Random();
```

This statement creates a `Random` object in memory and gives it the name `randomNumbers`. (You can give the object any name that you wish, I simply chose `randomNumbers` for this one.) Once you've created a `Random` object, you can use it to call a value-returning method named `nextInt`, which returns a random integer number. A `Random` object's `nextInt` method is similar to the `random` library function that is discussed in your textbook. The following code snippet shows an example of how to use it:

```
int number;
number = randomNumbers.nextInt(10);
```

In this example we are passing the argument 10 to the `randomNumbers.nextInt` method. This causes the method to return a random integer in the range of 0 through 9. The argument that we pass to the method is the upper limit of the range, but is not included in the range. In the code snippet we are assigning the random number to the `number` variable. Here is an example of how we would generate a random number in the range of 1 through 100:

```
int number;
number = randomNumbers.nextInt(100) + 1;
```

In this example we are passing the argument 100 to the `randomNumbers.nextInt` method. This generates a random integer in the range of 0 through 99. Then we add 1 to that number, thus causing it to be in the range of 1 through 100, and assignin the result to the `number` variable.

Program 6-1 shows a complete demonstration. This is the Java version of pseudocode Program 6-2 in your textbook. Let's take a closer look at the code:

- The statement in Line 9 creates a `Random` object in memory and gives it the name `randomNumbers`.
- Line 12 declares two `int` variables: `counter` and `number`. The `counter` variable will be used in a `for` loop, and the `number` variable will be used to hold random numbers.
- The `for` loop that begins in line 20 iterates 5 times.
- Inside the loop, the statement in line 22 generates a random number in the range of 0 through 100 and assigns it to the `number` variable.
- The statement in line 23 displays the value of the `number` variable.

Program 6-1

```
 1 import java.util.Random;
 2
 3 public class RandomNumbers
 4 {
 5    public static void main(String[] args)
 6    {
 7        // Create a Random object. This object will
 8        // provide us with the random numbers.
 9        Random randomNumbers = new Random();
10
11        // Declare variables.
12        int counter, number;
13
14        // The following loop displays five random
15        // numbers, each in the range of 1 through 100.
16        // When we call randomNumbers.nextInt(100) we get a
17        // random number in the range of 0 through 99. We add
18        // 1 to the value to adjust the range. As a result we
19        // get a random number in the range of 1 through 100.
20        for (counter = 1; counter <= 5; counter++)
21        {
22            number = randomNumbers.nextInt(100) + 1;
23            System.out.println(number);
24        }
25    }
26 }
```

This is the Java version of **Program 6-2** in your textbook!
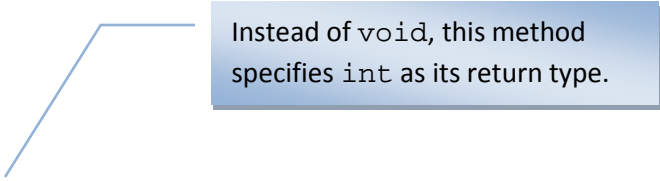
Program Output
```
57
50
26
86
44
```

Writing Your Own Value-Returning Methods

Up to now, all of the methods that you have written have been `void` methods, which means that they do not return a value. Recall that the key word `void` appears in a method header, as shown here:

```
public static void displayMessage()
```

This is the header for a method named `displayMessage`. Because it is a `void` method, it works like the modules that we discussed in Chapter 3 of your textbook. It is simply a procedure that executes when it is called, and does not return any value back to the statement that called it.

You can also write your own value-returning methods in Java. When you are writing a value-returning method, you must decide what type of value the method will return. This is because, instead of specifying `void` in the method header, you must specify the data type of the return value. A value-returning method will use `int`, `double`, `String`, `boolean`, or any other valid data type in its header. Here is an example of a method that returns an `int` value:

> Instead of `void`, this method specifies `int` as its return type.

```
public static int sum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    return result;
}
```

The name of this method is `sum`. Notice in the method header that instead of the word `void` we have specified `int` as the return type. This code defines a method named `sum` that accepts two `int` arguments. The arguments are passed into the parameter variables `num1` and `num2`. Inside the method, a local variable, `result`, is declared. The parameter variables `num1` and `num2` are added, and their sum is assigned to the `result` variable. The last statement in the method is as follows:

```
return result;
```

This is a `return` statement. You must have a `return` statement in a value-returning method. It causes the method to end execution and it returns a value to the statement that called the method. In a value-returning method, the general format of the return statement is as follows:

```
return Expression;
```

Expression is the value to be returned. It can be any expression that has a value, such as a variable, literal, or mathematical expression. In this case, the `sum` method returns the value of the `result`

variable. Program 6-2 shows a complete Java program that demonstrates this method. The program is the Java version of pseudocode Program 6-6 in your textbook.

Program 6-2

```java
 1 import java.util.Scanner;
 2
 3 public class ValueReturningMethod
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Create a Scanner object for keyboard input.
 8       Scanner keyboard = new Scanner(System.in);
 9
10       // Local variables
11       int firstAge, secondAge, total;
12
13       // Get the user's age and the user's
14       // best friend's age.
15       System.out.print("Enter your age: ");
16       firstAge = keyboard.nextInt();
17       System.out.print("Enter your best friend's age: ");
18       secondAge = keyboard.nextInt();
19
20       // Get the sum of both ages.
21       total = sum(firstAge, secondAge);
22
23       // Display the sum.
24       System.out.print("Together you are " + total + " years old.");
25    }
26
27    // The sum function accepts two int arguments and
28    // returns the sum of those arguments as an int.
29    public static int sum(int num1, int num2)
30    {
31       int result;
32       result = num1 + num2;
33       return result;
34    }
35 }
```

> This is the Java version of **Program 6-6** in your textbook!

Program Output
```
Enter your age: 22 [Enter]
Enter your best friend's age: 24 [Enter]
Together you are 46 years old.
```

Returning Strings

The following code shows an example of how a method can return string. Notice that the method header specifies `String` as the return type. This method accepts two string arguments (a person's first name and last name). It combines those two strings into a string that contains the person's full name. The full name is then returned.

```
public static String fullName(String firstName, String lastName)
{
    String name;

    name = firstName + " " + lastName;
    return name;
}
```

The following code snippet shows how we might call the method:

```
String customerName;
customerName = fullName("John", "Martin");
```

After this code executes, the value of the `custerName` variable will be "John Martin".

Returning a `boolean` Value

Methods can also return `boolean` values. The following method accepts an argument and returns `true` if the argument is within the range of 1 through 100, or `false` otherwise:

```
public static boolean isValid(int number)
{
    boolean status;

    if (number >= 1 && number <= 100)
        status = true;
    else
        status = false;

    return status;
}
```

The following code shows an `if-else` statement that uses a call to the method:

```
int value = 20;

if (isValid(value))
    System.out.println("The value is within range.");
else
    System.out.println("The value is out of range.");
```

When this code executes, the message "The value is within range." will be displayed.

`Math` Methods

In Chapter 2 you were introduced to the `Math.pow` method, which returns the value of a number raised to a power. Table 6-1 describes several of the `Math` class's methods, including `Math.pow`.

Table 6-1 Several `Math` Class Methods

abs	*Example Usage:* `y = Math.abs(x);` *Description:* Returns the absolute value of the argument. This method can accept and return values of the `double`, `float`, `int`, and `long` data types.
acos	*Example Usage:* `y = Math.acos(x);` *Description:* Returns the arc-cosine of the argument. The argument should be the cosine of an angle. (The argument's value must be in the range from −1.0 through 1.0.) The return type and the argument are `doubles`.
asin	*Example Usage:* `y = Math.asin(x);` *Description:* Returns the arc-sine of the argument. The argument should be the sine of an angle. (The argument's value must be in the range from −1.0 through 1.0.) The return type and the argument are `doubles`.
atan	*Example Usage:* `y = Math.atan(x);` *Description:* Returns the arc-tangent of the argument. The argument should be the tangent of an angle. The return type and the argument are `doubles`.
ceil	*Example Usage:* `y = Math.ceil(x);` *Description:* Returns the smallest number that is greater than or equal to the argument. The return type and the argument are `doubles`.
cos	*Example Usage:* `y = Math.cos(x);` *Description:* Returns the cosine of the argument. The argument should be an angle expressed in radians. The return type and the argument are `doubles`.
exp	*Example Usage:* `y = Math.exp(x);` *Description:* Computes the exponential function of the argument, which is e^x. The return type and the argument are `doubles`.

`floor`	*Example Usage:* `y = Math.floor(x);` *Description:* Returns the largest number that is less than or equal to the argument. The return type and the argument are `doubles`.
`log`	*Example Usage:* `y = Math.log(x);` *Description:* Returns the natural logarithm of the argument. The return type and the argument are `doubles`.
`pow`	*Example Usage:* `y = Math.pow(x, z);` *Description:* Returns the value of the first argument raised to the power of the second argument. The return type and the argument are `doubles`.
`round`	*Example Usage:* `y = Math.round(x);` *Description:* Returns the value of the argument, as an integer, rounded to the nearest whole number. The argument is expected to be a `double` or a `float`. If the argument is a `double`, the return type is `long`. If the argument is a `float`, the return type is `int`.
`sin`	*Example Usage:* `y = Math.sin(x);` *Description:* Returns the sine of the argument. The argument should be an angle expressed in radians. The return type and the argument are `doubles`.
`sqrt`	*Example Usage:* `y = Math.sqrt(x);` *Description:* Returns the square root of the argument. The return type and argument are `doubles`.
`tan`	*Example Usage:* `y = Math.tan(x);` *Description:* Returns the tangent of the argument. The argument should be an angle expressed in radians. The return type and the argument are `doubles`.

toDegrees	*Example Usage:* `y = Math.toDegrees(x);` *Description:* Accepts as an argument an angle in radians. The angle converted to degrees is returned. The argument and return values are both `doubles`.
toRadians	*Example Usage:* `y = Math.toRadians(x);` *Description:* Accepts as an argument an angle in degrees. The angle converted to radians is returned. The argument and return values are both `doubles`.

Data Type Conversion

Chapter 6 in your textbook discusses the `toReal` and `toInteger` functions as pseudocode approaches for converting a string to an integer or a real value. In Java you use the `Double.parseDouble` and `Integer.parseInt` methods for these purposes. Table 6-2 summarizes the methods.

Table 6-2 Data Type Conversion Methods

Method	Description	Example
`Double.parseDouble`	Convert a string to a `double`.	`double num;` `num = Double.parseDouble(str);`
`Integer.parseInt`	Convert a string to an `int`.	`int num;` `num = Integer.parseInt(str);`

Formatting Floating-Point Numbers

In your Java programs you can create a `DecimalFormat` object, and use that object to format the appearance of floating-point values. For example, if you want to display a floating-point value as a currency amount, you will want to round that value to two decimal places, and insert commas where needed.

First, you need to write the following statement at the top of your program:

```
import java.util.DecimalFormat;
```

Then you create a `DecimalFormat` object in the following manner:

```
DecimalFormat formatter = new DecimalFormat("#,##0.00");
```

This statement creates a `DecimalFormat` object in memory and gives it the name `formatter`. Notice the string that appears inside the parentheses. This is known as a format pattern, and it uses special characters to control the way that the object will format floating-point numbers. In this example the string `"#,##0.00"` is being used as the formatting pattern. This formatting pattern will cause numbers to be formatted with two digits after the decimal point, at least one digit before the decimal point, and will cause commas to be printed in the number if it is large enough. This is the formatting pattern that you want to use for currency amounts.

After you have created the `DecimalFormat` object, you can use it to call the `format` method. Here is an example:

```
DecimalFormat formatter = new DecimalFormat("#,##0.00");
double amount = 12477.27699;
System.out.println(formatter.format(amount));
```

This code snippet will display the following:

```
12,477.28
```

Notice that this is not formatted exactly the way that the pseudocode function currencyFormat does in your textbook because the DecimalFormat object does not include a dollar sign. You will have to explicitly display that, as shown here:

```
DecimalFormat formatter = new DecimalFormat("#,##0.00");
double amount = 12477.27699;
System.out.println("$" + formatter.format(amount));
```

This code snippet will display the following:

```
$12,477.28
```

String Methods

Getting a String's length

The following code snippet shows an example of how you get the length of a string in Java:

```
// Declare and initialize a string variable.
String name = "Charlemagne";

// Assign the length of the string to the strlen variable.
int strlen = name.length();
```

This code declares a `String` variable named `name`, and initializes it with the string `"Charlemagne"`. Then, it declares an `int` variable named `strlen`. The `strlen` variable is initialized with the value returned from the `name.length()` method. This method returns the length of the string stored in name. In this case, the value 11 will be returned from the method.

Appending a String to Another String

Appending a string to another string is called concatenation. In Java you can perform string concatenation in two ways: using the + operator, and using the `concat` method. Here is an example of how the + operator works:

```
String lastName = "Conway";
String salutation = "Mr. ";
String properName;
properName = salutation + lastName;
```

After this code executes the `properName` variable will contain the string `"Mr. Conway"`. Here is an example of how you perform the same operation using the `concat` method:

```
String lastName = "Conway";
String salutation = "Mr. ";
String properName;
properName = salutation.concat(lastName);
```

The last statement in this code snippet calls the `salutation.concat` method, passing `lastName` as an argument. The method will return a copy of the string in the `salutation` variable, with the contents of the `lastName` variable concatenated to it. After this code executes the `properName` variable will contain the string `"Mr. Conway"`.

The `toUpperCase` and `toLowerCase` Methods

The `toUpperCase` method returns a copy of a string with all of its letters converted to uppercase. Here is an example:

```
String littleName = "herman";
String bigName = littleName.toUpperCase();
```

After this code executes, the `bigName` variable will hold the string "HERMAN".

The `toLowerCase` method returns a copy of a string with all of its letters converted to lowercase. Here is an example:

```
String bigName = "HERMAN";
String littleName = bigName.toLowerCase();
```

After this code executes, the `littleName` variable will hold the string "herman".

The `substring` Method

The `substring` method returns part of another string. (A string that is part of another string is commonly referred to as a "substring.") The first argument specifies the substring's starting position and the second argument specifies the substring's ending position. The character at the starting position is included in the substring, but the character at the ending position is not. Here is an example of how the method is used:

```
String fullName = "Cynthia Susan Lee";
String middleName = fullName.substring(8, 13);
System.out.println("The full name is " + fullName);
System.out.println("The middle name is " + middleName);
```

The code will produce the following output:

```
The full name is Cynthia Susan Lee
The middle name is Susan
```

The `indexOf` Method

In Java you can use the `indexOf` method to perform a task similar to that of the `contains` function discussed in your textbook. The `indexOf` methods searches for substrings within a string. Here is the general format:

string1`.indexOf(`*string2*`, `*start*`)`

In the general format *string1* and *string2* are strings, and *start* is an integer. The search begins at the position passed into *start* and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, –1 is returned. The following code shows an example. It displays the starting positions of each occurrence of the word "and" within a string.

```
String str = "and a one and a two and a three";
int position;

System.out.println("The word and appears at the " +
                   "following locations.");

position = str.indexOf("and");

while (position != -1)
{
   System.out.println(position);
   position = str.indexOf("and", position + 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
0
10
20
```

Chapter 7   Input Validation

Chapter 7 in your textbook discusses the process of input validation in detail. There are no new language features introduced in the chapter, so here we will simply show you a Java version of the pseudocode Program 7-2. This program uses an input validation loop in lines 42 through 47 to validate that the value entered by the user is not negative.

Program 7-1

```java
1 import java.util.Scanner;
2
3 public class InputValidation
4 {
5     public static void main(String[] args)
6     {
7         // Create a Scanner object for keyboard input.
8         Scanner keyboard = new Scanner(System.in);
9
10        // Local variable
11        String doAnother;
12
13        do
14        {
15            // Calculate and display a retail price.
16            showRetail();
17
18            // Do this again?
19            System.out.print("Do you have another item? (Enter y for yes): ");
20            doAnother = keyboard.next();
21        } while (doAnother.equals("y") || doAnother.equals("Y"));
22    }
23
24    // The showRetail module gets an item's wholesale cost
25    // from the user and displays its retail price.
26    public static void showRetail()
27    {
28        // Create a Scanner object for keyboard input.
29        Scanner keyboard = new Scanner(System.in);
30
31        // Local variables
32        double wholesale, retail;
33
34        // Constant for the markup percentage
35        final double MARKUP = 2.5;
36
37        // Get the wholesale cost.
38        System.out.print("Enter an item's wholesale cost: ");
39        wholesale = keyboard.nextDouble();
40
41        // Validate the wholesale cost.
42        while (wholesale < 0)
43        {
44            System.out.println("The cost cannot be negative. Please");
45            System.out.print("enter the correct wholesale cost: ");
46            wholesale = keyboard.nextDouble();
47        }
48
49        // Calculate the retail price.
50        retail = wholesale * MARKUP;
```

> This is the Java version of **Program 7-2** in your textbook!

```
51
52        // Display the retail price.
53        System.out.println("The retail price is $" + retail);
54    }
55 }
```

Program Output

Enter an item's wholesale cost: **-1 [*Enter*]**
The cost cannot be negative. Please
enter the correct wholesale cost: **1.50 [*Enter*]**
The retail price is $3.75
Do you have another item? (Enter y for yes): **n [*Enter*]**

Chapter 8   Arrays

Here is an example of an array declaration in Java:

```java
int[] numbers = new int[6];
```

This statement declares `numbers` as an `int` array. The size declarator specifies that the array has 6 elements. As mentioned in your textbook, it is a good practice to use a named constant for the size declarator, as shown here:

```java
final int SIZE = 6;
int[] numbers = new int[SIZE];
```

Here is another example:

```java
final int SIZE = 200;
double[] temperatures = new double[SIZE];
```

This code snippet declares `temperatures` as an array of 200 `doubles`. Here is one more:

```java
final int SIZE = 10;
String[] names = new String[SIZE];
```

This declares `names` as an array of 10 `Strings`.

Array Elements and Subscripts

You access each element of an array with a subscript. As discussed in your textbook, the first element's subscript is 0, the second element's subscript is 1, and so forth. The last element's subscript is the array size minus 1. Program 8-1 shows an example of an array being used to hold values entered by the user. This is the Java version of pseudocode Program 8-1 in your textbook.

Program 8-1
```java
 1 import java.util.Scanner;
 2
 3 public class ArrayInput
 4 {
 5     public static void main(String[] args)
 6     {
 7         // Create a Scanner object for keyboard input.
 8         Scanner keyboard = new Scanner(System.in);
 9
10         // Create a constant for the number of employees.
11         int SIZE = 3;
12
13         // Declare an array to hold the number of hours
14         // worked by each employee.
15         int[] hours = new int[SIZE];
```

This is the Java version of **Program 8-1** in your textbook!

```
16
17        // Get the hours worked by employee 1.
18        System.out.print("Enter the hours worked by employee 1: ");
19        hours[0] = keyboard.nextInt();
20
21        // Get the hours worked by employee 2.
22        System.out.print("Enter the hours worked by employee 2: ");
23        hours[1] = keyboard.nextInt();
24
25        // Get the hours worked by employee 3.
26        System.out.print("Enter the hours worked by employee 3: ");
27        hours[2] = keyboard.nextInt();
28
29        // Display the values entered.
30        System.out.println("The hours you entered are:");
31        System.out.println(hours[0]);
32        System.out.println(hours[1]);
33        System.out.println(hours[2]);
34    }
35 }
```

Program Output

```
Enter the hours worked by employee 1: 40 [Enter]
Enter the hours worked by employee 2: 20 [Enter]
Enter the hours worked by employee 3: 15 [Enter]
The hours you entered are:
40
20
15
```

The `length` Attribute

Each array in Java has an attribute named `length`. The value of the attribute is the number of elements in the array. For example, consider the array created by the following statement:

```
double[] temperatures = new double[25];
```

Because the `temperatures` array has 25 elements, the following statement would assign 25 to the variable `size`:

```
size = temperatures.length;
```

The `length` attribute can be useful when processing the entire contents of an array. For example, the following loop steps through an array and displays the contents of each element. The array's `length` attribute is used in the test expression as the upper limit for the loop control variable:

```
for (int i = 0; i < temperatures.length; i++)
    System.out.println(temperatures[i]);
```

Be careful not to cause an off-by-one error when using the `length` attribute as the upper limit of a subscript. The `length` attribute contains the number of elements in an array. The largest subscript in an array is `length − 1`.

Using a Loop to Process an Array

It is usually much more efficient to use a loop to access an array's elements, rather than writing separate statements to access each element. Program 8-2 demonstrates how to use a loop to step through an array's elements. This is the Java version of pseudocode Program 8-3 in your textbook.

Program 8-2

```java
 1 import java.util.Scanner;
 2
 3 public class ArrayLoop
 4 {
 5     public static void main(String[] args)
 6     {
 7         // Create a Scanner object for keyboard input.
 8         Scanner keyboard = new Scanner(System.in);
 9
10         // Create a constant for the number of employees.
11         final int SIZE = 3;
12
13         // Declare an array to hold the number of hours
14         // worked by each employee.
15         int[] hours = new int[SIZE];
16
17         // Declare a variable to use in the loops.
18         int index;
19
20         // Get the hours for each employee.
21         for (index = 0; index <= SIZE - 1; index++)
22         {
23             System.out.print("Enter the hours worked by " +
24                              "employee number " + (index + 1) +
25                              ": " );
26             hours[index] = keyboard.nextInt();
27         }
28
29         // Display the values entered.
30         for (index = 0; index <= SIZE - 1; index++)
31             System.out.println(hours[index]);
32     }
33 }
```

This is the Java version of **Program 8-3** in your textbook!

Program Output
```
Enter the hours worked by employee 1: 40 [Enter]
Enter the hours worked by employee 2: 20 [Enter]
Enter the hours worked by employee 3: 15 [Enter]
40
```

```
20
15
```

Initializing an Array

You can initialize an array with values when you declare it. Here is an example:

```
int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

This statement declares days as an array of `ints`, and stores initial values in the array. The series of values inside the braces and separated with commas is called an initialization list. These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days[0]`, the second value, 28, is stored in `days[1]`, and so forth.) Note that you do not use the `new` key word when you use an initialization list. Java automatically creates the array and stores the values in the initialization list in it.

The Java compiler determines the size of the array by the number of items in the initialization list. Because there are 12 items in the example statement's initialization list, the array will have 12 elements.

Sequentially Searching an Array

Section 8.2 in your textbook discusses the sequential search algorithm, in which a program steps through each of an array's elements searching for a specific value. Program 8-3 shows an example of the sequential search algorithm. This is the Java version of pseudocode Program 8-6 in the textbook.

Program 8-3
```
 1 public class SequentialSearch
 2 {                                          This is the Java version of
 3    public static void main(String[] args)  Program 8-6 in your textbook!
 4    {
 5       // Declare an array to hold test scores.
 6       int[] scores = { 87, 75, 98, 100, 82,
 7                        72, 88, 92, 60,  78 };
 8
 9       // Declare a Boolean variable to act as a flag.
10       boolean found;
11
12       // Declare a variable to use as a loop counter.
13       int index;
14
15       // The flag must initially be set to False.
16       found = false;
17
18       // Set the counter variable to 0.
19       index = 0;
20
```

```
21          // Step through the array searching for a
22          // score equal to 100.
23          while (found == false && index < scores.length)
24          {
25              if (scores[index] == 100)
26                  found = true;
27              else
28                  index = index + 1;
29          }
30
31          // Display the search results.
32          if (found)
33              System.out.println("You earned 100 on test number " +
34                                  (index + 1));
35          else
36              System.out.println("You did not earn 100 on any test.");
37      }
38 }
```

Program Output
```
You earned 100 on test number 4
```

Searching a String Array

Program 8-4 demonstrates how to find a string in a string array. This is the Java version of pseudocode Program 8-7 in the textbook.

Program 8-4
```
1 import java.util.Scanner;
2
3 public class StringArraySearch
4 {
5      public static void main(String[] args)
6      {
7          // Create a Scanner object for keyboard input.
8          Scanner keyboard = new Scanner(System.in);
9
10         // Declare a constant for the array size.
11         final int SIZE = 6;
12
13         // Declare a String array initialized with values.
14         String[] names = { "Ava Fischer", "Chris Rich",
15                            "Geri Rose", "Matt Hoyle",
16                            "Rose Harrison", "Giovanni Ricci" };
17
18         // Declare a variable to hold the search value.
19         String searchValue;
20
21         // Declare a Boolean variable to act as a flag.
22         boolean found;
23
24         // Declare a counter variable for the array.
```

This is the Java version of **Program 8-7** in your textbook!

```
25        int index;
26
27        // The flag must initally be set to False.
28        found = false;
29
30        // Set the counter variable to 0.
31        index = 0;
32
33        // Get the string to search for.
34        System.out.print("Enter a name to search for in the array: ");
35        searchValue = keyboard.nextLine();
36
37        // Step through the array searching for
38        // the specified name.
39        while (found == false && index < names.length)
40        {
41            if (names[index].equals(searchValue))
42                found = true;
43          else
44                index = index + 1;
45        }
46
47        // Display the search results.
48        if (found)
49            System.out.println("That name was found in element " + index);
50        else
51            System.out.println("That name was not found in the array.");
52    }
53 }
```

Program Output

```
Enter a name to search for in the array: Matt Hoyle [Enter]
That name was found in element 3
```

Program Output

```
Enter a name to search for in the array: Terry Thompson [Enter]
That name was not found in the array.
```

Passing an Array as an Argument to a Method

When passing an array as an argument to a method in Java, it is not necessary to pass a separate argument indicating the array's size. This is because arrays in Java have the `length` attribute that reports the array's size. The following code shows a method that has been written to accept an array as an argument:

```
public static void showArray(int[] array)
{
    for (int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
}
```

Notice that the parameter variable, `array`, is declared as an `int` array. When we call this method we must pass an `int` array to it as an argument. Assuming that `numbers` is the name of an `int` array, here is an example of a method call that passes the `numbers` array as an argument to the `showArray` method:

```
showArray(numbers);
```

Program 8-5 gives a complete demonstration of passing an array to a method. This is the Java version of pseudocode Program 8-13 in your textbook.

Program 8-5
```
 1 public class PassArray
 2 {
 3     public static void main(String[] args)
 4     {
 5         // A constant for the array size
 6         final int SIZE = 5;
 7
 8         // An array initialized with values
 9         int[] numbers = { 2, 4, 6, 8, 10 };
10
11         // A variable to hold the sum of the elements
12         int sum;
13
14         // Get the sum of the elements.
15         sum = getTotal(numbers);
16
17         // Display the sum of the array elements.
18         System.out.println("The sum of the array elements is " + sum);
19     }
20
21     // The getTotal function accepts an Integer array, and the
22     // array's size as arguments. It returns the total of the
23     // array elements.
24     public static int getTotal(int[] array)
25     {
26         // Loop counter
27         int index;
28
29         // Accumulator, initialized to 0
30         int total = 0;
31
32         // Get each test score.
33         for (index = 0; index < array.length; index++)
34         {
35             total = total + array[index];
36         }
37
```

This is the Java version of **Program 8-13** in your textbook!

```
38        return total;
39    }
40 }
```
Program Output
```
The sum of the array elements is 30
```

Two-Dimensional Arrays

Here is an example declaration of a two-dimensional array with three rows and four columns:

```
double[][] scores = new double[3][4];
```

The two sets of brackets in the data type indicate that the scores variable will reference a two-dimensional array. The numbers 3 and 4 are size declarators. The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice that each size declarator is enclosed in its own set of brackets.

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `scores` array, the elements in row 0 are referenced as follows:

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

The elements in row 1 are as follows:

```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

And the elements in row 2 are as follows:

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

To access one of the elements in a two-dimensional array, you must use both subscripts. For example, the following statement stores the number 95 in `scores[2][1]`:

```
scores[2][1] = 95;
```

Programs that process two-dimensional arrays can do so with nested loops. For example, the following code prompts the user to enter a score, once for each element in the array:

```
final int ROWS = 3;
final int COLS = 4;
double[][] scores = new double[ROWS][COLS];
for (int row = 0; row < ROWS; row++)
{
    for (int col = 0; col < COLS; col++)
    {
        System.out.print("Enter a score: ");
        scores[row][col] = keyboard.nextDouble();
    }
}
```

And the following code displays all the elements in the `scores` array:

```
for (int row = 0; row < ROWS; row++)
{
    for (int col = 0; col < COLS; col++)
    {
        System.out.println(scores[row][col]);
    }
}
```

Program 8-6 shows a complete example. It declares an array with three rows and four columns, prompts the user for values to store in each element, and then displays the values in each element. This is the Java example of pseudocode Program 8-16 in your textbook.

Program 8-5
```
 1  import java.util.Scanner;
 2
 3  public class TwoDArray
 4  {
 5      public static void main(String[] args)
 6      {
 7          // Create a Scanner object for keyboard input.
 8          Scanner keyboard = new Scanner(System.in);
 9
10          // Create a 2D array
11          final int ROWS = 3;
12          final int COLS = 4;
13          int[][] values = new int[ROWS][COLS];
14
15          // Counter variables for rows and columns
16          int row, col;
17
18          // Get values to store in the array.
19          for (row = 0; row <= ROWS - 1; row++)
20          {
21              for (col = 0; col <= COLS - 1; col++)
22              {
23                  System.out.print("Enter a number: ");
24                  values[row][col] = keyboard.nextInt();
```

> This is the Java version of **Program 8-16** in your textbook!

```
25          }
26        }
27
28        // Display the values in the array.
29        System.out.println("Here are the values you entered.");
30        for (row = 0; row <= ROWS - 1; row++)
31        {
32            for (col = 0; col <= COLS - 1; col++)
33            {
34                System.out.println(values[row][col]);
35            }
36        }
37    }
38 }
```

Program Output

```
Enter a number: 1 [Enter]
Enter a number: 2 [Enter]
Enter a number: 3 [Enter]
Enter a number: 4 [Enter]
Enter a number: 5 [Enter]
Enter a number: 6 [Enter]
Enter a number: 7 [Enter]
Enter a number: 8 [Enter]
Enter a number: 9 [Enter]
Enter a number: 10 [Enter]
Enter a number: 11 [Enter]
Enter a number: 12 [Enter]
Here are the values you entered.
1
2
3
4
5
6
7
8
9
10
11
12
```

Arrays with Three or More Dimensions

Java allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array declaration:

```
double[][][] seats = new double[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

Chapter 9   Sorting and Searching Arrays

Chapter 9 discusses the following sorting algorithms:

- Bubble Sort
- Selection Sort
- Insertion Sort

The Binary Search algorithm is also discussed. The textbook chapter examines these algorithms in detail, and no new language features are introduced. For these reasons we will simply present the Java code for the algorithms in this chapter. For more in-depth coverage of the logic involved, consult the textbook.

Bubble Sort

Program 9-1 is only a partial program. It shows the Java version of pseudocode Program 9-1, which is the Bubble Sort algorithm.

Program 9-1

```
 1 public class BubbleSortAlgorithm
 2 {
 3    //   Note: This is not a complete program.
 4    //
 5    //   The bubbleSort method uses the bubble sort algorithm
 6    //   to sort an int array.
 7    //   Note the following:
 8    //   (1) We do not have to pass the array size because in
 9    //       Java arrays have a length field.
10    //   (2) We do not have a separate method to swap values.
11    //       This is because Java does not allow pass by
12    //       reference. The swap is perfomed inside this method.
13
14    public static void bubbleSort(int[] array)
15    {
16       int maxElement;  // Marks the last element to compare
17       int index;       // Index of an element to compare
18       int temp;        // Used to swap to elements
19
20       // The outer loop positions maxElement at the last element
21       // to compare during each pass through the array. Initially
22       // maxElement is the index of the last element in the array.
23       // During each iteration, it is decreased by one.
24       for (maxElement = array.length - 1; maxElement >= 0; maxElement--)
25       {
26          // The inner loop steps through the array, comparing
27          // each element with its neighbor. All of the elements
28          // from index 0 thrugh maxElement are involved in the
29          // comparison. If two elements are out of order, they
30          // are swapped.
31          for (index = 0; index <= maxElement - 1; index++)
32          {
33             // Compare an element with its neighbor.
```

This is the Java version of **Program 9-1** in your textbook!

```
34              if (array[index] > array[index + 1])
35              {
36                 // Swap the two elements.
37                 temp = array[index];
38                 array[index] = array[index + 1];
39                 array[index + 1] = temp;
40              }
41           }
42        }
43     }
44 }
```

Selection Sort

Program 9-2 is also a partial program. It shows the Java version of the `selectionSort` pseudocode module that is shown in Program 9-5 in your textbook.

This is the Java version of the `selectionSort` Module shown in **Program 9-5** in your textbook!

Program 9-2
```
 1 public class SelectionSortAlgorithm
 2 {
 3    //    Note: This is not a complete program.
 4    //
 5    //    The selectionSort method performs a selection sort on an
 6    //    int array. The array is sorted in ascending order.
 7
 8    public static void selectionSort(int[] array)
 9    {
10       int startScan;    // Starting position of the scan
11       int index;        // To hold a subscript value
12       int minIndex;     // Element with smallest value in the scan
13       int minValue;     // The smallest value found in the scan
14
15       // The outer loop iterates once for each element in the
16       // array. The startScan variable marks the position where
17       // the scan should begin.
18       for (startScan = 0; startScan < (array.length-1); startScan++)
19       {
20          // Assume the first element in the scannable area
21          // is the smallest value.
22          minIndex = startScan;
23          minValue = array[startScan];
24
25          // Scan the array, starting at the 2nd element in
26          // the scannable area. We are looking for the smallest
27          // value in the scannable area.
28          for(index = startScan + 1; index < array.length; index++)
29          {
30             if (array[index] < minValue)
31             {
32                minValue = array[index];
33                minIndex = index;
34             }
35          }
36
37          // Swap the element with the smallest value
```

```
38            // with the first element in the scannable area.
39            array[minIndex] = array[startScan];
40            array[startScan] = minValue;
41         }
42      }
43 }
```

Insertion Sort

Program 9-3 is also a partial program. It shows the Java version of the `insertionSort` pseudocode module that is shown in Program 9-6 in your textbook.

Program 9-3

> This is the Java version of the `insertionSort` Module shown in **Program 9-6** in your textbook!

```
 1 public class InsertionSortAlgorithm
 2 {
 3     // Note: This is not a complete program.
 4     //
 5     // The insertionSort method performs an insertion sort on
 6     // an int array. The array is sorted in ascending order.
 7
 8     public static void insertionSort(int[] array)
 9     {
10         int unsortedValue;  // The first unsorted value
11         int scan;           // Used to scan the array
12
13         // The outer loop steps the index variable through
14         // each subscript in the array, starting at 1. This
15         // is because element 0 is considered already sorted.
16         for (int index = 1; index < array.length; index++)
17         {
18             // The first element outside the sorted subset is
19             // array[index]. Store the value of this element
20             // in unsortedValue.
21             unsortedValue = array[index];
22
23             // Start scan at the subscript of the first element
24             // outside the sorted subset.
25             scan = index;
26
27             // Move the first element outside the sorted subset
28             // into its proper position within the sorted subset.
29             while (scan > 0 && array[scan-1] > unsortedValue)
30             {
31                 array[scan] = array[scan - 1];
32                 scan--;
33             }
34
35             // Insert the unsorted value in its proper position
36             // within the sorted subset.
37             array[scan] = unsortedValue;
38         }
```

```
39     }
40 }
```

Binary Search

Program 9-4 is also a partial program. It shows the Java version of the `binarySearch` pseudocode module that is shown in Program 9-7 in your textbook.

Program 9-4

This is the Java version of the `binarySearch` Module shown in **Program 9-7** in your textbook!

```
 1 public class BinarySearchAlgorithm
 2 {
 3    //   Note: This is not a complete program.
 4    //
 5    //   The binarySearch method performs a binary search on a
 6    //   String array. The array is searched for the string passed
 7    //   to value. If the string is found, its array subscript
 8    //   is returned. Otherwise, -1 is returned indicating the
 9    //   value was not found in the array.
10
11    public static int binarySearch(String[] array, String value)
12    {
13       int first;        // First array element
14       int last;         // Last array element
15       int middle;       // Mid point of search
16       int position;     // Position of search value
17       boolean found;    // Flag
18
19       // Set the inital values.
20       first = 0;
21       last = array.length - 1;
22       position = -1;
23       found = false;
24
25       // Search for the value.
26       while (!found && first <= last)
27       {
28          // Calculate mid point
29          middle = (first + last) / 2;
30
31          // If value is found at midpoint...
32          if (array[middle].equals(value))
33          {
34             found = true;
35             position = middle;
36          }
37          // else if value is in lower half...
38          else if (array[middle].compareTo(value) > 0)
39             last = middle - 1;
40          // else if value is in upper half....
41          else
42             first = middle + 1;
43       }
44
```

```
45        // Return the position of the item, or -1
46        // if it was not found.
47        return position;
48    }
49 }
```

Chapter 10   Files

Opening a File and Writing Data to It

To work with files in a Java program you first write the following statement at the top of your program:

```
import java.io.*;
```

Then, in the method where you wish to open a file and write data to it you will create a `PrintWriter` object. Here is an example of a statement that creates a `PrintWriter` object:

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```

This statement does the following:

- It creates a `PrintWriter` object in memory. The `PrintWriter` object's name is `outputFile`. You will use this `PrintWriter` object in your program to manage the file.
- It opens a file on the disk named `StudentData.txt`.

After this statement has executed, we will be able to use the `PrintWriter` object named `outputFile` to write data to the `StudentData.txt` file. You can think of it this way: In memory we have a `PrintWriter` object that we refer to in our code as `outputFile`. That object is connected to a file on the disk named `StudentData.txt`. If we want to write data to the `StudentData.txt` file, we will use the `PrintWriter` object. (Note that if the `StudentData.txt` file does not exist, this statement will create the file. If the file already exists, its contents will be erased. Either way, after this statement executes an empty file named `StudentData.txt` will exist on the disk.)

Once you have created `PrintWriter` object and opened a file, you can write data to the file using the `println` methods. You already know how to use `println` with `System.out` to display data on the screen. It is used the same way with a `PrintWriter` object to write data to a file. For example, assuming that `outputFile` is a `PrintWriter` object, the following statement writes the string `"Jim"` to the file:

```
outputFile.println("Jim");
```

Assuming that `payRate` is a variable, the following statement writes the value of the `payRate` variable to the file:

```
outputFile.println(payRate);
```

Closing a File

When the program is finished writing data to the file, it must close the file. Assuming that `outputFile` is the name of a `PrintWriter` object, here is an example of how to call the `close` method to close the file:

```
outputFile.close();
```

Once a file is closed, the connection between it and the `PrintWriter` object is removed. In order to perform further operations on the file, it must be opened again.

Program 10-1 demonstrates how to create a `PrintWriter` object (and open a file for output), write some data to the file, and close the file. This is the Java version of pseudocode Program 10-1 in your textbook.

Program 10-1
```
 1 import java.io.*;                                          Don't Forget this!
 2
 3 public class FileWriteDemo
 4 {
 5     public static void main(String[] args) throws IOException
 6     {
 7         // Declare a PrintWriter variable named myFile and open a
 8         // file named philosphers.dat.
 9         PrintWriter myFile = new PrintWriter("philosophers.dat");
10
11         // Write the names of three philosophers to the file.
12         myFile.println("John Locke");
13         myFile.println("David Hume");
14         myFile.println("Edmund Burke");
15
16         // Close the file.                    This is the Java version of
17         myFile.close();                       Program 10-1 in your textbook!
18     }
19 }
```

Notice in line 5, the header for the `main` method ends with the clause `throws IOException`. This is required because of Java's advanced error handling mechanism. You do not need to be concerned about the details of the clause, but you do need to remember that it is required for any method that uses the file techniques shown in this book.
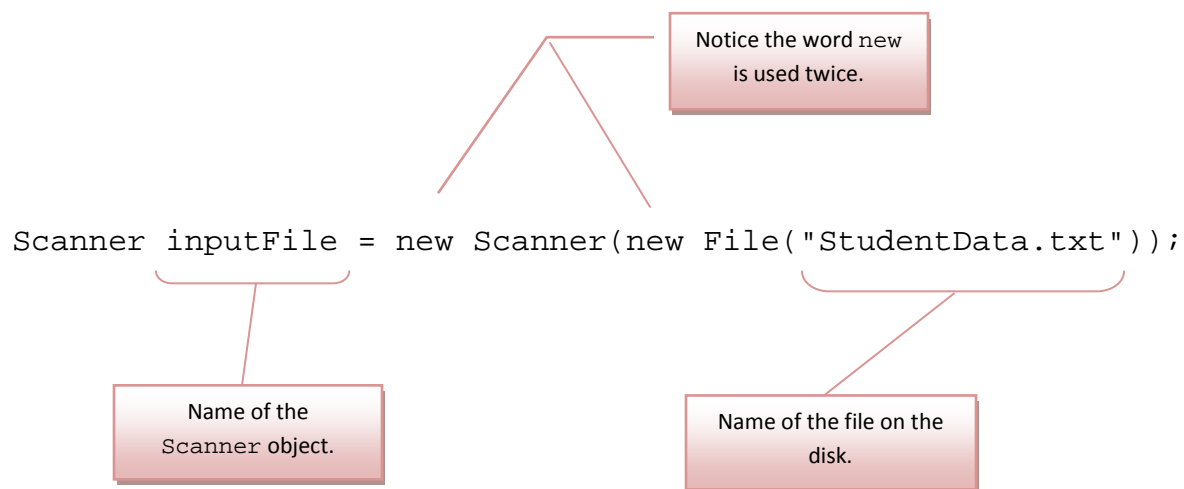
When this program executes, line 9 creates a file named philosophers.dat on the disk, and lines 12 through 14 write the strings "John Locke", "David Hume", and "Edmund Burke" to the file. Line 17 closes the file.

Opening a File and Reading Data from It

In Chapter 2 you learned how to create a `Scanner` object to read input from the keyboard. You can also use a `Scanner` object to read data from a file. First, you need this statement at the top of your program:

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

Then, inside the method that needs to read data from a file you create a `Scanner` object and connect it to that file. Here is an example:

Notice the word `new` is used twice.

```
Scanner inputFile = new Scanner(new File("StudentData.txt"));
```

Name of the `Scanner` object.

Name of the file on the disk.

This statement creates a `Scanner` object in memory. The name of the `Scanner` object is `inputFile`. The file `StudentData.txt` is opened for reading, and the `Scanner` object is connected to it. After this statement executes, you will be able to use the `Scanner` object to read data from the file.

Once you've connected a `Scanner` object to a file, you can use the `nextLine` method to read a string from the file, the `nextInt` method to read an integer, or the `nextDouble` method to read a `double`.

Program 10-2 shows an example that reads strings from a file. This program opens the `philosophers.dat` file that was created by Program 10-1. This is the Java version of pseudocode Program 10-2 in your textbook. Here are some specific points about the program:

- Line 14 creates a `Scanner` object named `myFile`, opens a file on the disk named `philosophers.dat`, and connects the `Scanner` object to the file.
- Line 18 reads a line of text from the file and assigns it to the `name1` variable.
- Line 19 reads the next line of text from the file and assigns it to the `name2` variable.

- Line 20 reads the next line of text from the file and assigns it to the `name3` variable.
- Line 29 closes the file.

Program 10-2

```
 1 import java.io.*;
 2 import java.util.Scanner;
 3
 4 public class FileReadDemo
 5 {
 6     public static void main(String[] args) throws IOException
 7     {
 8         // Declare three variables that will hold the values
 9         // read from the file.
10         String name1, name2, name3;
11
12         // Declare a Scanner variable named myFile and open a
13         // file named philosphers.dat.
14         Scanner myFile = new Scanner(new File("philosophers.dat"));
15
16         // Read the names of three philosophers from the file
17         // into the variables.
18         name1 = myFile.nextLine();
19         name2 = myFile.nextLine();
20         name3 = myFile.nextLine();
21
22         // Display the names that were read.
23         System.out.println("Here are the names of three philosophers:");
24         System.out.println(name1);
25         System.out.println(name2);
26         System.out.println(name3);
27
28         // Close the file.
29         myFile.close();
30     }
31 }
```

Don't Forget this!

This is the Java version of **Program 10-2** in your textbook!

Program Output

```
Here are the names of three philosophers:
John Locke
David Hume
Edmund Burke
```

Appending Data to an Existing File

You learned earlier that when you create a `PrintWiter` object and open a file, if the file already exists its contents will be erased. Sometimes you want to open an existing file for writing, and preserve its contents so new data can be appended to the existing contents. When this is the case, you create a `PrintWriter` object with a statement like this:

```
PrintWriter outputFile =
   new PrintWriter(new FileWriter("MyFriends.txt", true));
```

This statement opens an existing file named `MyFriends.txt`. The contents of the file are not erased, and when we use the `println` method to write data to the file, the data will be appended to the end of the file.

Using Loops to Process Files

Program 10-3 demonstrates how a loop can be used to collect items of data to be stored in a file. This is the Java version of pseudocode Program 10-3 in your textbook.

Program 10-3

```java
 1 import java.io.*;
 2 import java.util.Scanner;
 3
 4 public class BuildSalesFile
 5 {
 6    public static void main(String[] args) throws IOException
 7    {
 8       // Variable to hold the number of days.
 9       int numDays;
10
11       // Counter variable for the loop.
12       int counter;
13
14       // Variable to hold an amount of sales.
15       double sales;
16
17       // Create a Scanner object for keyboard input.
18       Scanner keyboard = new Scanner(System.in);
19
20       // Get the number of days.
21       System.out.print("For how many days do you have sales? ");
22       numDays = keyboard.nextInt();
23
24       // Open a file named sales.dat.
25       PrintWriter salesFile = new PrintWriter("sales.dat");
26
27       // Get the amount of sales for each day and write
28       // it to the file.
29       for (counter = 1; counter <= numDays; counter++)
30       {
31          // Get the sales for a day.
32          System.out.print("Enter the sales for day #" + counter + ": ");
33          sales = keyboard.nextDouble();
34
35          // Write the amount to the file.
36          salesFile.println(sales);
37       }
38
39       // Close the file.
40       salesFile.close();
41       System.out.println("Data written to sales.dat.");
42    }
43 }
```

This is the Java version of **Program 10-3** in your textbook!

```
For how many days do you have sales? 5 [Enter]
Enter the sales for day #1: 1000 [Enter]
Enter the sales for day #2: 2000 [Enter]
Enter the sales for day #3: 3000 [Enter]
Enter the sales for day #4: 4000 [Enter]
Enter the sales for day #5: 5000 [Enter]
Data written to sales.dat.
```

Detecting the End of a File

Sometimes you need to read a file's contents, and you do not know the number of items that are stored in the file. When this is the case, you can use the `Scanner` method `hasNext` to determine whether the file contains another item before you attempt to read an item from it. If there is more data that can be read from the file, the `hasNext` method returns `true`. If the end of the file has been reached and there is no more data to read, the `hasNext` method returns `false`.

Program 10-4 demonstrates how to use the `hasNext` method. This is the Java version of pseudocode Program 10-4 in your textbook. The program opens the sales.dat file that was created by Program 10-3. It reads and displays each item of data in the file.

Program 10-4

```
 1 import java.io.*;
 2 import java.util.Scanner;
 3
 4 public class ReadSalesFile
 5 {
 6     public static void main(String[] args) throws IOException
 7     {
 8         // Variable to hold an amount of sales.
 9         double sales;
10
11         // Open a file named sales.dat.
12         Scanner salesFile = new Scanner(new File("sales.dat"));
13
14         System.out.println("Here are the sales amounts:");
15
16         // Read all of the items in the file and display them.
17         while (salesFile.hasNext())
18         {
19             sales = salesFile.nextDouble();
20             System.out.printf("$%,.2f\n", sales);
21         }
22
23         // Close the file.
24         salesFile.close();
```

This is the Java version of **Program 10-4** in your textbook!

```
25    }
26 }
```

Program Output

```
Here are the sales amounts:
$1,000.00
$2,000.00
$3,000.00
$4,000.00
$5,000.00
```

Chapter 11   Menu-Driven Programs

Chapter 11 in your textbook discusses menu-driven programs. A menu-driven program presents a list of operations that the user may select from (the menu), and then performs the operation that the user selected. There are no new language features introduced in the chapter, so here we will simply show you a Java program that is menu-driven. Program 11-1 is the Java version of the pseudocode Program 11-3.

Program 11-1

```java
 1 import java.util.Scanner;
 2
 3 public class MenuDriven
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Declare a variable to hold the
 8       // user's menu selection.
 9       int menuSelection;
10
11       // Declare variables to hold the units
12       // of measurement.
13       double inches, centimeters, feet, meters,
14              miles, kilometers;
15
16       // Create a Scanner object for keyboard input.
17       Scanner keyboard = new Scanner(System.in);
18
19       // Display the menu.
20       System.out.println("1. Convert inches to centimeters.");
21       System.out.println("2. Convert feet to meters.");
22       System.out.println("3. Convert miles to kilometers.");
23       System.out.println();
24
25       // Prompt the user for a selection
26       System.out.print("Enter your selection: ");
27       menuSelection = keyboard.nextInt();
28
29       // Validate the menu selection.
30       while (menuSelection < 1 || menuSelection > 3)
31       {
32          System.out.println("That is an invalid selection.");
33          System.out.print("Enter 1, 2, or 3: ");
34          menuSelection = keyboard.nextInt();
35       }
36
37       // Perform the selected operation.
38       switch(menuSelection)
```

This is the Java version of **Program 11-3** in your textbook!

```
39          {
40              case 1:
41                  // Convert inches to centimeters.
42                  System.out.print("Enter the number of inches: ");
43                  inches = keyboard.nextDouble();
44                  centimeters = inches * 2.54;
45                  System.out.println("That is equal to " + centimeters +
46                                      " centimeters.");
47                  break;
48
49              case 2:
50                  // Convert feet to meters.
51                  System.out.print("Enter the number of feet: ");
52                  feet = keyboard.nextDouble();
53                  meters = feet * 0.3048;
54                  System.out.println("That is equal to " + meters +
55                                      " meters.");
56                  break;
57
58              case 3:
59                  // Convert miles to kilometers.
60                  System.out.print("Enter the number of miles: ");
61                  miles = keyboard.nextDouble();
62                  kilometers = miles * 1.609;
63                  System.out.println("That is equal to " + kilometers +
64                                      " kilometers.");
65                  break;
66          }
67      }
68 }
```

Program Output
```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 1 [Enter]
Enter the number of inches: 10 [Enter]
That is equal to 25.4 centimeters.
```

Program Output
```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 2 [Enter]
Enter the number of feet: 10 [Enter]
```

```
That is equal to 3.048 meters.
```

Program Output
```
1. Convert inches to centimeters.
2. Convert feet to meters.
3. Convert miles to kilometers.

Enter your selection: 4 [Enter]
That is an invalid selection.
Enter 1, 2, or 3: 3 [Enter]
Enter the number of miles: 10 [Enter]
That is equal to 16.09 kilometers.
```

Chapter 12   Text Processing

Character-By-Character Text Processing

In Java, strings are immutable objects, which means that once they have been created in memory, their value cannot be changed. This limits some of the text processing that can be done with String variables. As an alternative to the String variable, Java allows you to create StringBuilder objects. You can store a string in a StringBuilder object, and it allows you to directly modify the individual characters that it contains. In this chapter we will be working with StringBuilder objects.

Here is an example of how you create a StringBuilder object:

> The name of the StringBuilder object.

> The string to store in the StringBuilder object.

```
StringBuilder cityName = new StringBuilder("Charleston");
```

This statement creates a StringBuilder object named cityName. The object contains the string "Charleston". We can use this StringBuilder object to manipulate the individual characters of the string.

You use the StringBuilder method charAt to access a character at a specific location. For example, suppose you have the following declaration:

```
StringBuilder name = new StringBuilder("Jacob");
```
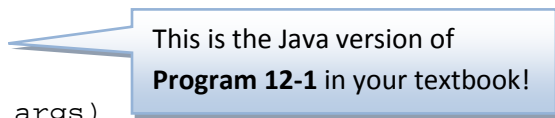
The following expression returns the first character in the string:

```
name.charAt(0)
```

When you call the StringBuilder method charAt, you pass a position number as an argument (the position numbers begin at 0) and the method returns the character at that position. Program 12-1 shows an example. This is the Java version of pseudocode Program 12-1 in your textbook.

Program 12-1

> This is the Java version of **Program 12-1** in your textbook!

```
 1 public class DisplayCharacters
 2 {
 3    public static void main(String[] args)
 4    {
 5       // Declare and initialize a string.
 6       StringBuilder name = new StringBuilder("Jacob");
 7
 8       // Display the individual characters in the string.
 9       System.out.println(name.charAt(0));
10       System.out.println(name.charAt(1));
11       System.out.println(name.charAt(2));
```

```
12          System.out.println(name.charAt(3));
13          System.out.println(name.charAt(4));
14     }
15 }
```
Program Output
```
J
a
c
o
b
```

Program 12-2 is the Java version of pseudocode Program 12-1 in your textbook. This program uses a loop to step through all of the characters in a `StringBuilder` object.

Program 12-2
```
 1 public class LoopDisplayCharacters
 2 {
 3     public static void main(String[] args)
 4     {
 5         // Declare and initialize a string.
 6         StringBuilder name = new StringBuilder("Jacob");
 7
 8         // Declare a variable to step through the string.
 9         int index;
10
11         // Display the individual characters in the string.
12         for (index = 0; index < name.length(); index++)
13             System.out.println(name.charAt(index));
14     }
15 }
```

This is the Java version of **Program 12-2** in your textbook!

Program Output
```
J
a
c
o
b
```

Changing the Value of a Specific Character

The `StringBuilder` method `charAt` returns the value of a character at a specific location. If you want to change the value of a specific character, you must use the `setCharAt` method. The `setCharAt` method takes two arguments: the position of the character that you want to change, and the character that you want to change it to. Program 12-3 shows an example. This is the Java version of pseudocode Program 12-3 in your textbook.

Program 12-3

```java
 1 import java.util.Scanner;
 2
 3 public class ChangeCharacters
 4 {
 5    public static void main(String[] args)
 6    {
 7       // Create a Scanner object for keyboard input.
 8       Scanner keyboard = new Scanner(System.in);
 9
10       // Declare a string to hold input.
11       String input;
12
13       // Declare a variable to step through the string.
14       int index;
15
16       // Prompt the user to enter a sentence.
17       System.out.print("Enter a sentence: ");
18       input = keyboard.nextLine();
19
20       // Store the string in a StringBuilder object.
21       StringBuilder str = new StringBuilder(input);
22
23       // Change each "t" to a "d".
24       for (index = 0; index < str.length(); index++)
25       {
26          if (str.charAt(index) == 't')
27          {
28             str.setCharAt(index, 'd');
29          }
30       }
31
32       // Display the modified string.
33       System.out.println(str);
34    }
35 }
```

This is the Java version of **Program 12-3** in your textbook!

Program Output

```
Enter a sentence: Look at that kitty cat! [Enter]
Look ad dhad kiddy cad!
```

The loop in lines 24 through 30 steps through all of the characters stored in the StringBuilder object. Inside the loop, the if statement that starts in line 26 determines whether the character at the position specified by index is 't'. If it is, the statement in line 28 changes that character to 'd'.

Character Literals in Java

You probably noticed that in line 26 we are using single quotes around the character literal 't', and in line 28 we are using single quotes around the character literal 'd'. In Java, there is a difference between a string literal and a character literal. String literals are encloses in double quotes, and character literals are enclosed in single quotes. If you are writing a relational expression, using one of the relational operators such as == , and the operand on the left side is a character, then the operand on the right side must also be a character or an error will occur. In line 26, the charAt method returns a character, so we have to compare it to the character literal 't'. If we mistakenly compare it to the

string literal `"t"` an error will occur. In line 28 we call the `setCharAt` method. It requires that the second argument be a character, so we pass the character literal `'d'`. If we mistakenly pass the string literal `"d"` an error will occur.

Character Testing Methods

Java provides methods that are similar to the character testing library functions shown in Table 12-2 in your textbook. The Java methods that are similar to those functions are shown here, in Table 12-1.

Table 12-1 Character Testing Methods

Function	Description
`Character.isDigit(character)`	Returns `True` if *character* is a numeric digit, or `False` otherwise.
`Character.isLetter(character)`	Returns `True` if *character* is an alphabetic letter or `False` otherwise.
`Character.isLowerCase(character)`	Returns `True` if *character* is a lowercase letter or `False` otherwise.
`Character.isUpperCase(character)`	Returns `True` if *character* is an uppercase letter or `False` otherwise.
`Character.isWhiteSpace(character)`	Returns `True` if *character* is a whitespace character or `False` otherwise. (A whitespace character is a space, a tab, or a newline.)

Program 12-4 demonstrates how the `Character.isUpperCase` method is used. This program is the Java version of Program 12-4 in your textbook.

Program 12-4

```
 1 import java.util.Scanner;
 2
 3 public class UpperCaseCounter
 4 {
 5     public static void main(String[] args)
 6     {
 7         // Create a Scanner object for keyboard input.
 8         Scanner keyboard = new Scanner(System.in);
 9
10         // Declare a string to hold input.
11         String input;
12
13         // Declare a variable to step through the string.
14         int index;
15
16         // Declare an accumulator variable to keep count
17         // of the number of uppercase letters.
18         int upperCaseCount = 0;
19
20         // Prompt the user to enter a sentence.
21         System.out.print("Enter a sentence: ");
22         input = keyboard.nextLine();
23
```

This is the Java version of **Program 12-4** in your textbook!

```
24          // Store the string in a StringBuilder object.
25          StringBuilder str = new StringBuilder(input);
26
27          // Count the number of uppercase letters.
28          for (index = 0; index < str.length(); index++)
29          {
30              if (Character.isUpperCase(str.charAt(index)))
31              {
32                  upperCaseCount = upperCaseCount + 1;
33              }
34          }
35
36          // Display the number of uppercase characters.
37          System.out.println("That string has " + upperCaseCount +
38                             " uppercase letters.");
39      }
40 }
```
Program Output
```
Enter a sentence: Mr. Jones will arrive TODAY! [Enter]
That string has 7 uppercase letters.
```

Inserting and Deleting Characters in a `StringBuilder`

There are `StringBuilder` methods for inserting and deleting characters in a string. These methods are similar to the library modules that are shown in Table 12-3 in your textbook. The `StringBuilder` methods that are similar to those functions are shown here, in Table 12-2.

Table 12-2 StringBuilder Insertion and Deletion Methods

Function	Description
objectName.insert(*position*, *string*)	*objectName* is the name of a `StringBuilder` object, *position* is an `int`, and *string* is a `String`. The method inserts *string* into the `StringBuilder` object, beginning at *position*.
objectName.delete(*start*, *end*)	*objectName* is the name of a `StringBuilder` object, *start* is an `int`, and *end* is an `int`. The method deletes from the `StringBuilder` object all of the characters beginning at the position specified by*start*, and ending at the position specified by *end*. The character at the ending position is NOT included in the deletion.

Here is an example of how we might use the `insert` method:

```
StringBuilder str = new StringBuilder("New City");
str.insert(4, "York ");
System.out.println(str);
```

The second statement inserts the string `"York  "` into the `StringBuilder`, beginning at position 4. The characters that are currently in the `StringBuilder` beginning at position 4 are moved to the right. In memory, the `StringBuilder` is automatically expanded in size to accommodate the inserted characters. If these statements were a complete program and we ran it, we would see `New York City` displayed on the screen.

Here is an example of how we might use the `delete` method:

```
StringBuilder str = new StringBuilder("I ate 1000 blueberries!");
str.delete(8, 10);
System.out.println(str);
```

The second statement deletes the characters at positions 8 through 9 in the `StringBuilder`. It should be noted that the position specified by the second argument is not included in the deletion. (This differs from the way the `delete` library module is described in your textbook.)The characters that previously appeared beginning at position 10 are shifted left to occupy the space left by the deleted characters. If these statements were a complete program and we ran it, we would see `I ate 10 blueberries!` displayed on the screen.

Chapter 13   Recursion

A Java method can call itself recursively, allowing you to design algorithms that recursively solve a problem. Chapter 13 in your textbook describes recursion in detail, discusses problem solving with recursion, and provides several pseudocode examples. Other than the technique of a method recursively calling itself, no new language features are introduced. In this chapter we will present Java versions of two of the pseudocode programs that are shown in the textbook. Both of these programs work exactly as the algorithms are described in the textbook. Program 13-1 is the Java version of pseudocode Program 13-2.

Program 13-1

```
 1 public class RecursionDemo                       This is the Java version of
 2 {                                                 Program 13-2 in your textbook!
 3     public static void main(String[] args)
 4     {
 5         // By passing the argument 5 to the message method
 6         // we are telling it to display the message 5 times.
 7         message(5);
 8     }
 9
10     public static void message(int n)
11     {
12         if (n > 0)
13         {
14             System.out.println("This is a recursive method.");
15             message(n - 1);
16         }
17     }
18 }
```
Program Output
```
This is a recursive method.
This is a recursive method.
This is a recursive method.
This is a recursive method.
This is a recursive method.
```

Next, Program 13-2 is the Java version of pseudocode Program 13-3. This program recursively calculates the factorial of a number.

Program 13-2

```
 1 import java.util.Scanner;                         This is the Java version of
 2                                                   Program 13-3 in your textbook!
 3 public class RecursiveFactorial
 4 {
 5     public static void main(String[] args)
 6     {
 7         int number;       // To hold a number
 8
```

```
 9          // Create a Scanner object for keyboard input.
10          Scanner keyboard = new Scanner(System.in);
11
12          // Get a number from the user.
13          System.out.print("Enter a non-negative integer: ");
14          number = keyboard.nextInt();
15
16          // Display the factorial of the number.
17          System.out.println("The factorial of " + number +
18                             " is " + factorial(number));
19      }
20
21      //   The factorial method uses recursion to calculate
22      //   the factorial of its argument, which is assumed
23      //   to be a nonnegative number.
24
25      private static int factorial(int n)
26      {
27         if (n == 0)
28            return 1;   // Base case
29         else
30            return n * factorial(n - 1);
31      }
32 }
```

Program Output

```
Enter a non-negative integer: 7 [Enter]
The factorial of 7 is 5040
```

Chapter 14   Object-Oriented Programming

Java is a powerful object-oriented language. An object is an entity that exists in the computer's memory while the program is running, in the same way that a variable exists in the computer's memory. An object, however, is much more powerful and useful than an ordinary variable. An object contains data and has the ability to perform operations on its data. An object's data is commonly referred to as the object's fields, and the operations that the object performs are the object's methods.

In the object-oriented way of programming, objects are used to perform many of the program's tasks. For example, in Java, you have used `Scanner` objects to read input from the keyboard, and data from a file. You have also used `DecimalFormat` objects to format floating-point numbers. And, you have used `StringBuilder` objects to manipulate strings.

In addition to the many objects that are provided by the Java language, you can create objects of your own design. The first step is to write a class. A class is like a blueprint. It is a declaration that specifies the fields and the methods for a particular type of object. When the program needs an object of that type, it creates an instance of the class. (An object is an instance of a class.)

Here is the general format of a class declaration in Java:

```
public class ClassName
{
      Field declarations and method definitions go here…
}
```

Chapter 14 in your textbook steps through the design of a `CellPhone` class. The following `CellPhone` class is the Java version of Class Listing 14-3. Notice that each of the field declarations (lines 4 through 6) begin with the key word `private`. This is an access specifier that makes the fields private to the class. No code outside the class can directly access private class fields. Also notice that each of the method headers begin with the `public` access specifier. This makes the methods public, which means that code outside the class can call the methods.

```
CellPhone Class (CellPhone.java)
 1 public class CellPhone
 2 {
 3     // Field declarations
 4     private String manufacturer;
 5     private String modelNumber;
 6     private double retailPrice;
 7
 8     // Method Definitions
 9     public void setManufacturer(String manufact)
10     {
11         manufacturer = manufact;
12     }
13
14     public void setModelNumber(String modNum)
15     {
```

This is the Java version of **Class Listing 14-3** in your textbook!

```
16          modelNumber = modNum;
17      }
18
19      public void setRetailPrice(double retail)
20      {
21          retailPrice = retail;
22      }
23
24      public String getManufacturer()
25      {
26          return manufacturer;
27      }
28
29      public String getModelNumber()
30      {
31          return modelNumber;
32      }
33
34      public double getRetailPrice()
35      {
36          return retailPrice;
37      }
38  }
```

You will also notice that none of the field and method declarations contain the `static` key word. All of the fields and methods in this class are *non-static*. The more you work with object-oriented code, the more you will understand the difference between static and non-static code, but let's go over a quick summary.

First, let's discuss what happens when a class field is non-static. A non-static class field belongs to a specific object. For example, the `CellPhone` class has three fields: `manufacturer`, `modelNumber`, and `retailPrice`. Because these fields are non-static, they do not actually exist in memory until an instance of the `CellPhone` class is created. When an instance of the `CellPhone` class is created, its fields are created. Because non-static fields belong to a specific instance of the class, they are commonly referred to as *instance fields*.

Now let's discuss the difference between static and non-static methods. Static methods, such as the `main` method that you have written in every program so far, are general purpose methods that can simply be called in the program. They are defined in a class, but you do not have to create an instance of the class in order to use them. Non-static methods, however, are intended to work on an instance of the class in which they are defined. For this reason, non-static methods are usually referred to as *instance methods*. In order to call an instance method, an instance of the class in which the method is declared must exist.

For example, all of the methods that are declared in the `CellPhone` class are instance methods. They are meant to operate on the data that belongs to an instance of the `CellPhone` class. Before we can call any of the `CellPhone` class's methods, we must create an instance of the `CellPhone` class (in other words, we must create a `CellPhone` object). Then, we can call the `setManufacter` method to set the manufacturer for that object. We can call the `setModelNumber` method to set the model

number for that object. And, we can call the `setRetailPrice` method to set the retail price for that object. Likewise, we can call the `getManufacturer`, `getModelNumber`, and `getRetail` methods to get the manufacturer, model number, and retail price for that object.

Program 14-1 shows how to create an instance of the `CellPhone` class. This is the Java version of pseudocode Program 14-1.

Program 14-1

```
 1 public class CellPhoneObjectDemo
 2 {
 3     public static void main(String[] args)
 4     {
 5         // Declare a variable that can reference
 6         // a CellPhone object.
 7         CellPhone myPhone;
 8
 9         // The following statement creates an object
10         // using the CellPhone class as its blueprint.
11         // The myPhone variable will reference the object.
12         myPhone = new CellPhone();
13
14         // Store values in the object's fields.
15         myPhone.setManufacturer("Motorola");
16         myPhone.setModelNumber("M1000");
17         myPhone.setRetailPrice(199.99);
18
19         // Display the values stored in the fields.
20         System.out.println("The manufacturer is " +
21                             myPhone.getManufacturer());
22         System.out.println("The model number is " +
23                             myPhone.getModelNumber());
24         System.out.println("The retail price is " +
25                             myPhone.getRetailPrice());
26     }
27 }
```

This is the Java version of **Program 14-1** in your textbook!

Program Output
```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
```

The statement in line 7 declares a `CellPhone` variable name `myPhone`. As described in your textbook, this is a special type of variable that can be used to reference a `CellPhone` object. This statement does not, however, create a `CellPhone` object in memory. That is done in line 12. The expression on the right side of the = operator creates a new `CellPhone` object in memory, and the = operator assigns the object's memory address to the `myPhone` variable. As a result, we say that the `myPhone` variable references a `CellPhone` object.

We can then use the `myPhone` variable to perform operations with the object that it references. This is demonstrated in line 15, where we use the `myPhone` variable to call the `setManufacturer` method. The `"Motorola"` string that we pass as an argument is assigned to the object's `manufacturer` field. Likewise, line 16 uses the `myPhone` variable to call the `setModelNumber` method. That causes the string `"M1000"` to be assigned to the object's `modelNumber` field. And, line 17 uses the `myPhone` variable to call the `setRetailPrice` method. This causes the value 199.99 to be stored in the object's `retailPrice` field. The statements that appear in lines 20 through 25 get the values that are stored in the object's fields and displays them on the screen.

Constructors

A class constructor in Java is a method that has the same name as the class. The following is a version of the `CellPhone` class that has a constructor. This is the Java version of Class Listing 14-4 in your textbook. The constructor appears in lines 9 through 14.

`CellPhone Class`

```
 1 public class CellPhone
 2 {
 3     // Field declarations
 4     private String manufacturer;
 5     private String modelNumber;
 6     private double retailPrice;
 7
 8     // Constructor
 9     public CellPhone(String manufact, String modNum, double retail)
10     {
11         manufacturer = manufact;
12         modelNumber = modNum;
13         retailPrice = retail;
14     }
15
16     // Mutator methods
17     public void setManufacturer(String manufact)
18     {
19         manufacturer = manufact;
20     }
21
22     public void setModelNumber(String modNum)
23     {
24         modelNumber = modNum;
25     }
26
27     public void setRetailPrice(double retail)
28     {
29         retailPrice = retail;
30     }
31
32     // Accessor methods
33     public String getManufacturer()
34     {
35         return manufacturer;
36     }
37
```

This is the Java version of **Class Listing 14-4** in your textbook!

```
38      public String getModelNumber()
39      {
40          return modelNumber;
41      }
42
43      public double getRetailPrice()
44      {
45          return retailPrice;
46      }
47 }
```

Program 14-2 demonstrates how to create an instance of the class, passing arguments to the constructor. This is the Java version of pseudocode Program 14-2. In line 12, an instance of the CellPhone class is created and the arguments "Motorola", "M1000", and 199.99 are passed to the constructor.

Program 14-2
```
 1 public class ConstructorDemo
 2 {
 3     public static void main(String[] args)
 4     {
 5         // Declare a variable that can reference
 6         // a CellPhone object.
 7         CellPhone myPhone;
 8
 9         // The following statement creates an object
10         // and initializes its fields with the values
11         // passed to the constructor.
12         myPhone = new CellPhone("Motorola", "M1000", 199.99);
13
14         // Display the values stored in the fields.
15         System.out.println("The manufacturer is " +
16                            myPhone.getManufacturer());
17         System.out.println("The model number is " +
18                            myPhone.getModelNumber());
19         System.out.println("The retail price is " +
20                            myPhone.getRetailPrice());
21     }
22 }
```

> This is the Java version of **Program 14-2** in your textbook!

Program Output
```
The manufacturer is Motorola
The model number is M1000
The retail price is 199.99
```

Inheritance

The inheritance example discussed in your textbook starts with the `GradedActivity` class (see Class Listing 14-8), which is used as a superclass. The Java version of the class is shown here:

```
GradedActivity Class
 1 public class GradedActivity
 2 {
 3     // The score field holds a numeric score.
 4     private double score;
 5
 6     // Mutator
 7     public void setScore(double s)
 8     {
 9         score = s;
10     }
11
12     // Accessor
13     public double getScore()
14     {
15         return score;
16     }
17
18     // getGrade method
19     public String getGrade()
20     {
21         // Local variable to hold a grade.
22         String grade;
23
24         // Determine the grade.
25         if (score >= 90)
26             grade = "A";
27         else if (score >= 80)
28             grade = "B";
29         else if (score >= 70)
30             grade = "C";
31         else if (score >= 60)
32             grade = "D";
33         else
34             grade = "F";
35
36         // Return the grade.
37         return grade;
38     }
39 }
```

This is the Java version of **Class Listing 14-8** in your textbook!

A subclass of the `GradedActivity` class named `FinalExam` is shown in your textbook in Class Listing 14-9. The Java version of the `FinalExam` class is shown here:

FinalExam Class

```java
1 public class FinalExam extends GradedActivity
2 {
3    // Fields
4    private int numQuestions;
5    private double pointsEach;
6    private int numMissed;
7
8    // The constructor sets the number of
9    // questions on the exam and the number
10   // of questions missed.
11   public FinalExam(int questions, int missed)
12   {
13      // Local variable to hold the numeric score.
14      double numericScore;
15
16      // the numQuestions and numMissed fields.
17      numQuestions = questions;
18      numMissed = missed;
19
20      // Calculate the points for each question
21      // and the numeric score for this exam.
22      pointsEach = 100.0 / questions;
23      numericScore = 100.0 - (missed * pointsEach);
24
25      // Call the inherited setScore method to
26      // set the numeric score.
27      setScore(numericScore);
28   }
29
30   // Accessors
31   public double getPointsEach()
32   {
33      return pointsEach;
34   }
35
36   public int getNumMissed()
37   {
38      return numMissed;
39   }
40 }
```

This is the Java version of **Class Listing 14-9** in your textbook!

Notice that in line 1, in the class header, the `extends GradedActivity` clause specifies that the `FinalExam` class extends a superclass, the `GradedActivity` class. Program 14-3 demonstrates the class. This is the Java version of pseudocode Program 14-5 in your textbook.

Program 14-3

```java
1 import java.util.Scanner;
2
3 public class InheritanceDemo
4 {
```

This is the Java version of **Program 14-5** in your textbook!

```
 5     public static void main(String[] args)
 6     {
 7        // Variables to hold user input.
 8        int questions, missed;
 9
10        // Create a Scanner object for keyboard input.
11        Scanner keyboard = new Scanner(System.in);
12
13        // Class variable to reference a FinalExam object.
14        FinalExam exam;
15
16        // Prompt the user for the number of questions
17        // on the exam.
18        System.out.print("Enter the number of questions on the exam: ");
19        questions = keyboard.nextInt();
20
21        // Prompt the user for the number of questions
22        // missed by the student.
23        System.out.print("Enter the number of questions that the " +
24                         "student missed: ");
25        missed = keyboard.nextInt();
26
27        // Create a FinalExam object.
28        exam = new FinalExam(questions, missed);
29
30        // Display the test results.
31        System.out.println("Each question on the exam counts " +
32                 exam.getPointsEach() + " points.");
33        System.out.println("The exam score is " + exam.getScore());
34        System.out.println("The exam grade is " + exam.getGrade());
35     }
36 }
```

Program Output

```
Enter the number of questions on the exam: 20 [Enter]
Enter the number of questions that the student missed: 3 [Enter]
Each question on the exam counts 5.0 points.
The exam score is 85.0
The exam grade is B
```

Polymorphism

Your textbook presents a polymorphism demonstration that uses the `Animal` class shown in Class Listing 14-10 as a superclass. The Java version of that class is shown here:

`Animal` Class

> This is the Java version of **Class Listing 14-10** in your textbook!

```
1 public class Animal
2 {
3     // showSpecies method
4     public void showSpecies()
5     {
6         System.out.println("I'm just a regular animal.");
7     }
8
```

```
 9      // makeSound method
10      public void makeSound()
11      {
12          System.out.println("Grrrrrr");
13      }
14  }
```

The Dog class, which extends the Animal class, is shown in Class Listing 14-11 in your textbook. The Java version of the Dog class is shown here:

Dog Class

```
 1  public class Dog extends Animal
 2  {
 3      // showSpecies method
 4      public void showSpecies()
 5      {
 6          System.out.println("I'm a dog.");
 7      }
 8
 9      // makeSound method
10      public void makeSound()
11      {
12          System.out.println("Woof! Woof!");
13      }
14  }
```

This is the Java version of **Class Listing 14-11** in your textbook!

The Cat class, which also extends the Animal class, is shown in Class Listing 14-12 in your textbook. The Java version of the Cat class is shown here:

Cat Class

```
 1  public class Cat extends Animal
 2  {
 3      // showSpecies method
 4      public void showSpecies()
 5      {
 6          System.out.println("I'm a cat.");
 7      }
 8
 9      // makeSound method
10      public void makeSound()
11      {
12          System.out.println("Meow");
13      }
14  }
```

This is the Java version of **Class Listing 14-12** in your textbook!

Program 14-4, shown here, demonstrates the polymorphic behavior of these classes, as discussed in your textbook. This is the Java version of pseudocode Program 14-6.

Program 14-4

```java
 1 public class PolymorphismDemo
 2 {
 3     public static void main(String[] args)
 4     {
 5         // Declare three class variables.
 6         Animal myAnimal;
 7         Dog myDog;
 8         Cat myCat;
 9
10         // Create an Animal object, a Dog object,
11         // and a Cat object.
12         myAnimal = new Animal();
13         myDog = new Dog();
14         myCat = new Cat();
15
16         // Show info about an animal.
17         System.out.println("Here is info about an animal.");
18         showAnimalInfo(myAnimal);
19         System.out.println();
20
21         // Show info about a dog.
22         System.out.println("Here is info about a dog.");
23         showAnimalInfo(myDog);
24         System.out.println();
25
26         // Show info about a cat.
27         System.out.println("Here is info about a cat.");
28         showAnimalInfo(myCat);
29     }
30
31     // The showAnimalInfo method accepts an Animal
32     // object as an argument and displays information
33     // about it.
34     public static void showAnimalInfo(Animal creature)
35     {
36         creature.showSpecies();
37         creature.makeSound();
38     }
39 }
```

This is the Java version of **Program 14-6** in your textbook!

Program Output

```
Here is info about an animal.
I'm just a regular animal.
Grrrrr

Here is info about a dog.
I'm a dog.
Woof! Woof!

Here is info about a cat.
I'm a cat.
Meow
```

Chapter 15   GUI Applications and Event-Driven Programming

Chapter 15 discusses how, in some languages, GUI components can be created in an integrated development environment (IDE), and how event handlers can be written to respond to the user's interactions. The Java language does not come with an IDE, but Sun Microsystems provides a free IDE known as NetBeans. You can download NetBeans for free from www.netbeans.org/downloads. Be sure to download the Java version.

In this chapter we will provide a simple, step by step tutorial that leads you through the process of creating a simple Java GUI application in NetBeans. (This chapter was written with NetBeans version 6.5)

STEP 1: Start NetBeans. The Screen will appear similar to Figure 15-1. As shown in the figure, click the New Project Icon 🗂 to start a new project.

Figure 15-1 The NetBeans IDE
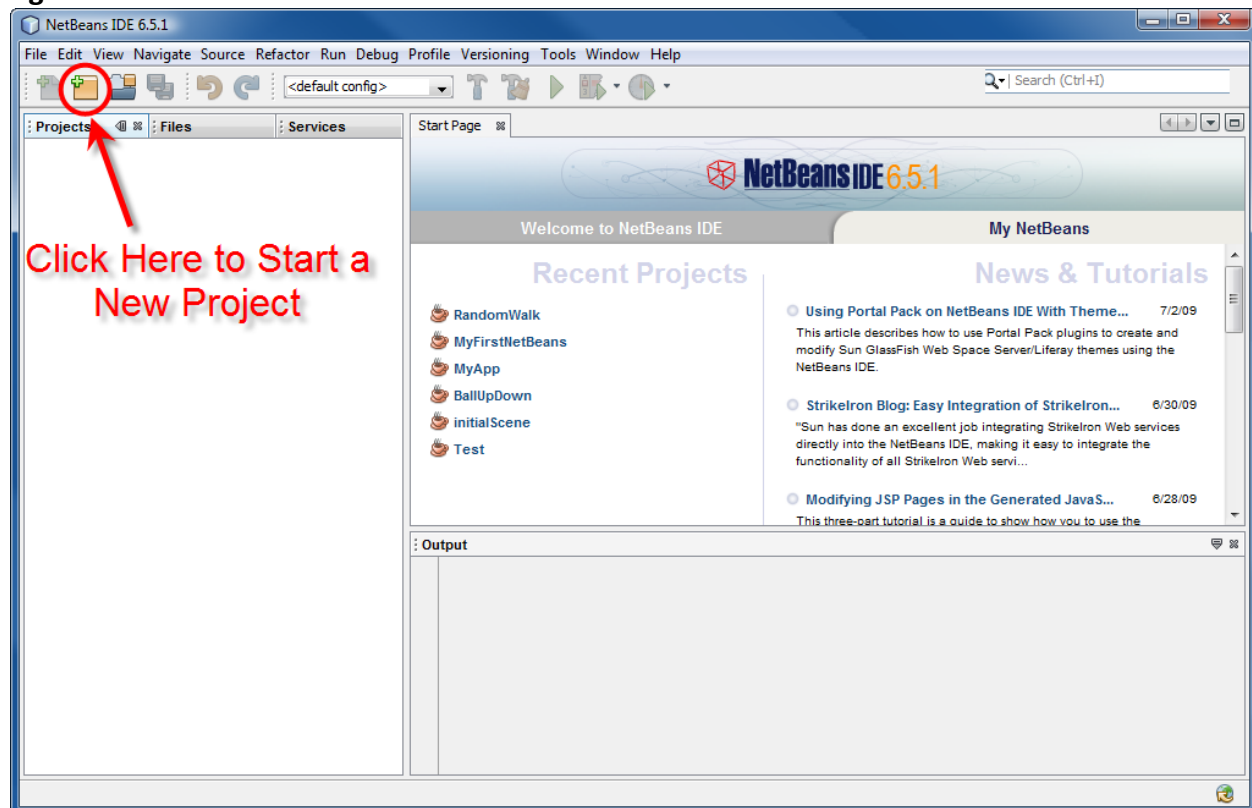
STEP 2: The New Project Window, shown in Figure 15-2, will appear next. As shown in the figure, select **Java** in the Categories pane and then select **Java Desktop Application** in the Projects pane. Then click the Next button at the bottom of the window.

Figure 15-2 The New Project Window
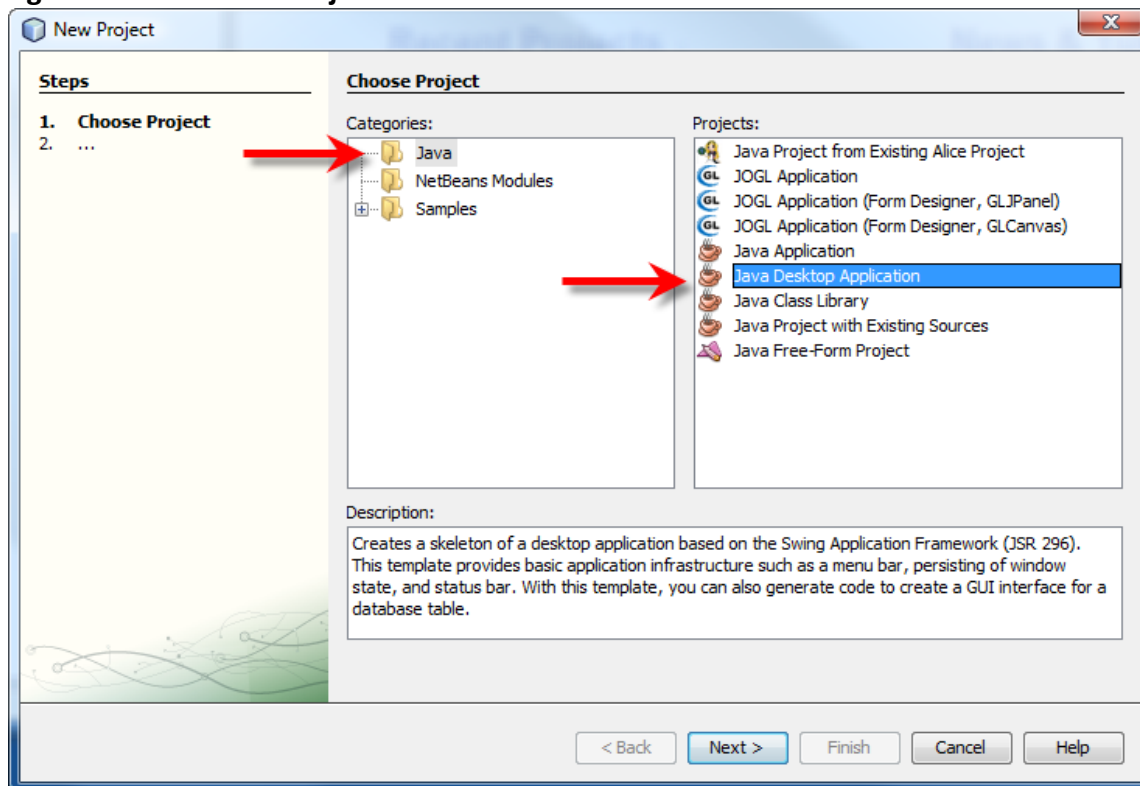
Figure 15-3 The New Desktop Application Window
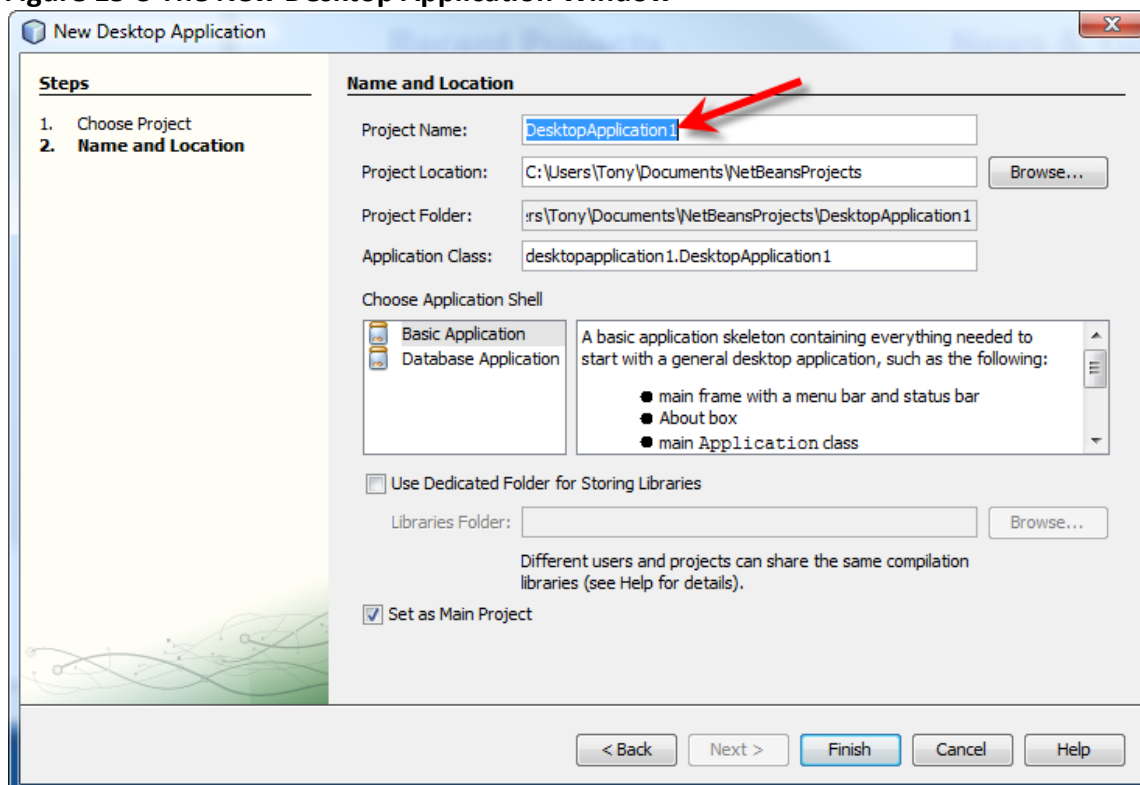
STEP 3: The New Desktop Application Window, shown in Figure 15-3, will appear next. Notice that under Name and Location, the default name for the project is DesktopApplication1 (or something similar to that on your system). Although you can keep this default name for your project, it is usually a good idea to change the name to something more meaningful. Change the project name to **MyFirstNetBeansApp** and then click the Finish button.

STEP 4: The screen should now appear similar to that shown in Figure 15-4. The figure points out three major areas of the screen:

- **The Design Area:** This is where you will visually place GUI components in your application's window. As shown in the figure, make sure the **Design** button at the top of this area is selected.
- **The Palette Window:** This window provides a set of GUI components that you can use in your application's GUI window.
- **The Properties Window:** This window allows you to change the properties of a GUI component to alter its appearance or its behavior.

Figure 15-4 NetBeans Screen

Make sure the Design button is selected.

Palette Window

Properties Window

Design Area
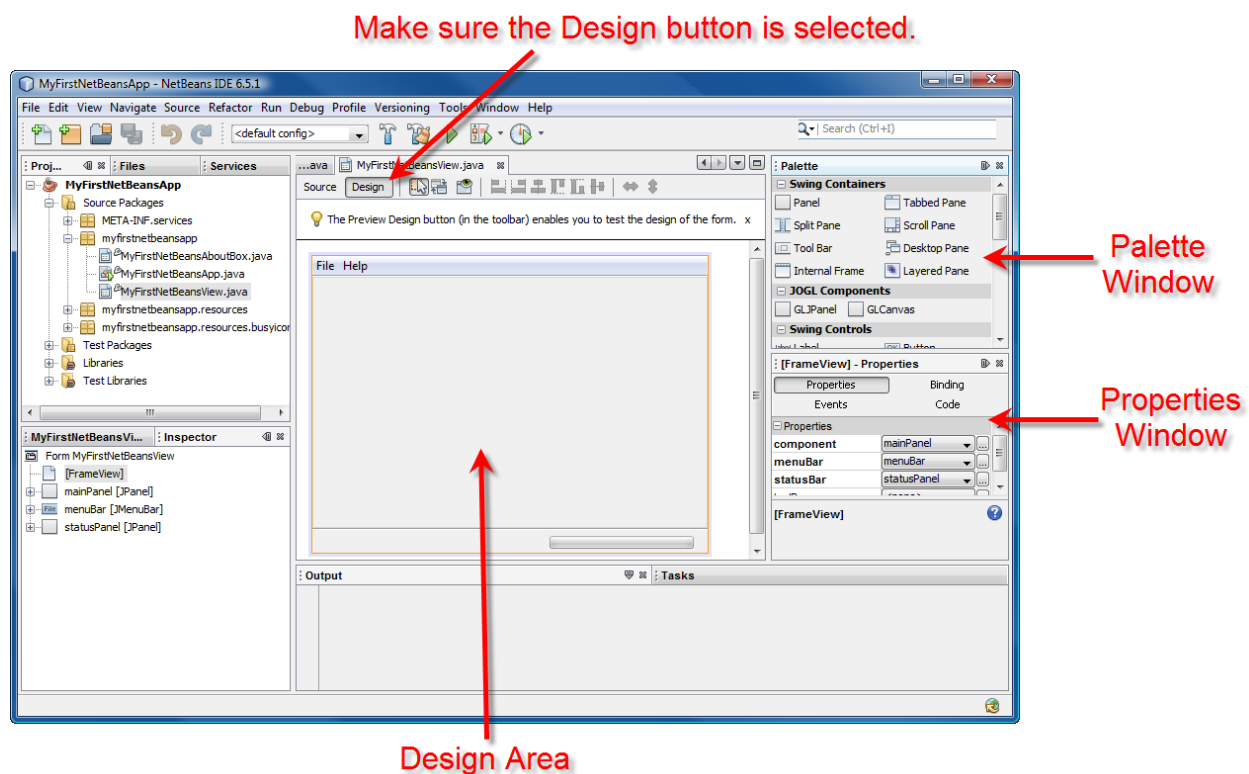
Note -- If the Palette Window is not visible, do one of the following:

- Press the Ctrl-Shift-8 keys, or…
- Click Window on the menu bar, then click Palette

Note -- If the Properties Window is not visible, do one of the following:

- Press the Ctrl-Shift-7 keys, or…
- Click Window on the menu bar, then click Properties

STEP 5: The components that are shown in the Palette window are grouped into categories. Scroll down in the Palette window until you see the **Swing Controls** category, as shown in Figure 15-5. Notice that one of the items shown in this category is the Label. Click and drag the Label control from the Palette window into the Design Area. Drop the control (by releasing the mouse button) at the approximate location shown in Figure 16-6.

Figure 15-5 The Swing Controls Category in the Palette Window
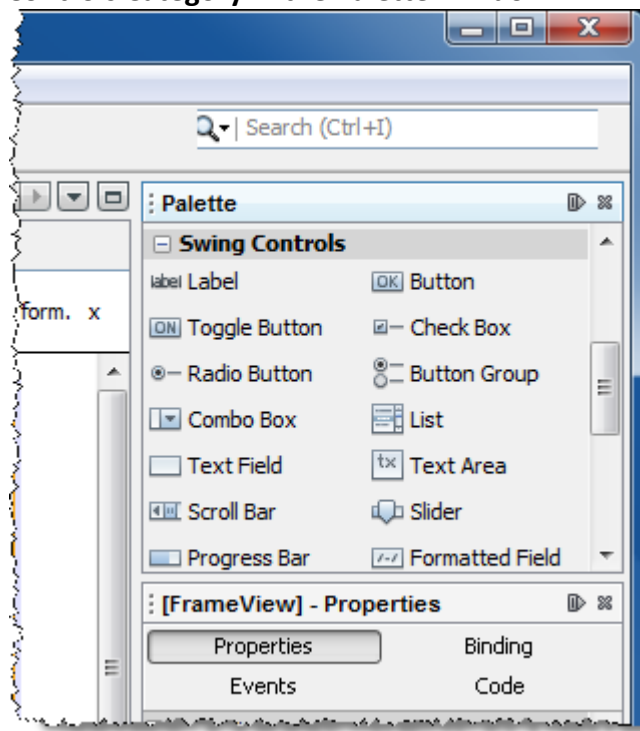
STEP 6: The Design Area should now appear something like Figure 16-7. Notice that the Label shows the name `jLabel1`. This is the name that the component will have in the application's Java code. Also notice that sizing handles are placed around the Label. (These are the little squares that appear around the label, as shown in Figure 15-7.) You can click and drag these sizing handles to resize the label. The sizing handles also indicate that the Label is currently selected.

> *Note* -- If you click anywhere in the Design Area outside the Label that you just placed, you will deselect the Label and the sizing handles will disappear. To select the Label again, just click it with the mouse and the sizing handles will reappear around it.

Figure 15-6 Click and Drag the Label Control into the Design Area
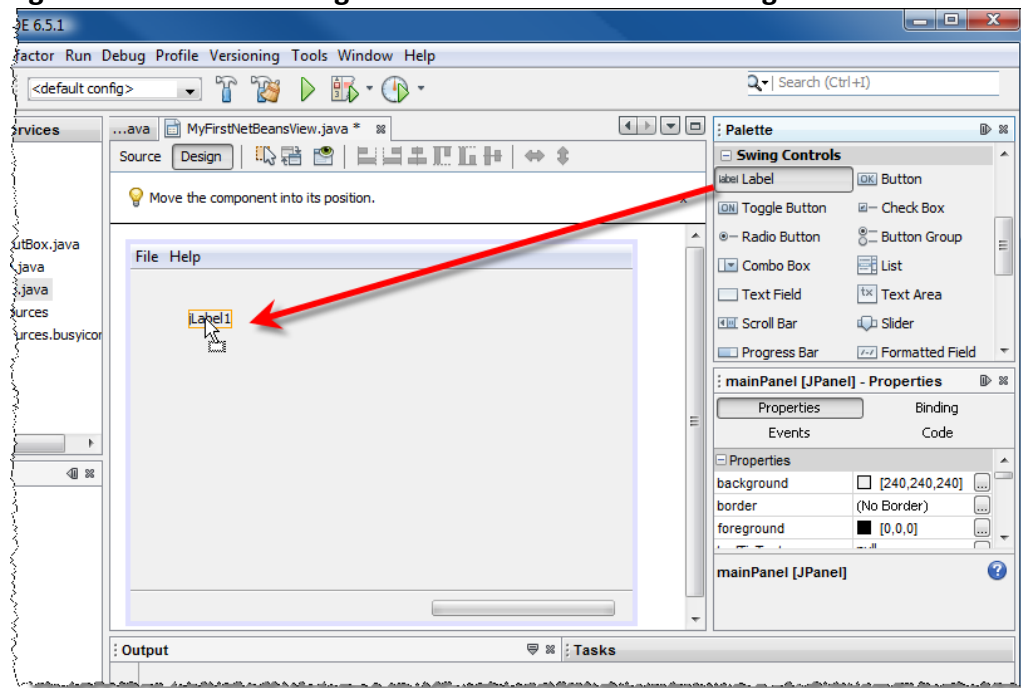
Figure 15-7 The Design Area with the Label Control Placed
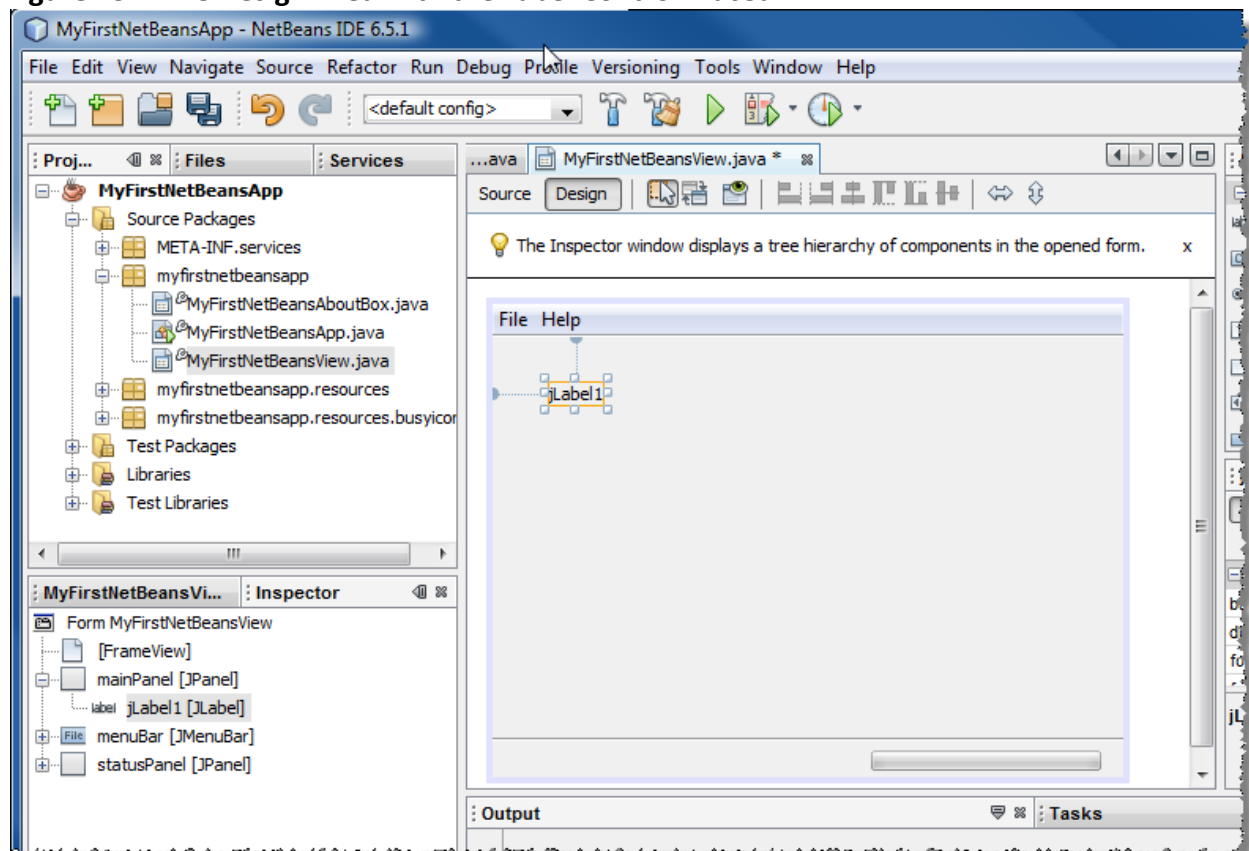
STEP 7: Now you will change the text that is displayed by the the JLabel1 Label that you just placed in the Design Area. Make sure the Label is selected, and do the following:

- Make sure the Properties button is selected in the Properties Window, as shown in Figure 15-8.
- Scroll down in the Properties Window until you find the text property, as shown in Figure 15-8. The text property controls the text that is displayed by the Label control. Currently the property's value is jLabel1.
- Double-click the property value, as indicated in Figure 15-8, type **Enter your name**, and press the Enter key.

After doing this, the Design area and the Label's text property should appear as shown in Figure 15-9.

> *Note* -- You have changed the text that the Label displays, but you have not changed the name of the Label component in the Java code. It is still named jLabel1.

Figure 15-8 Locate and Change the text Property
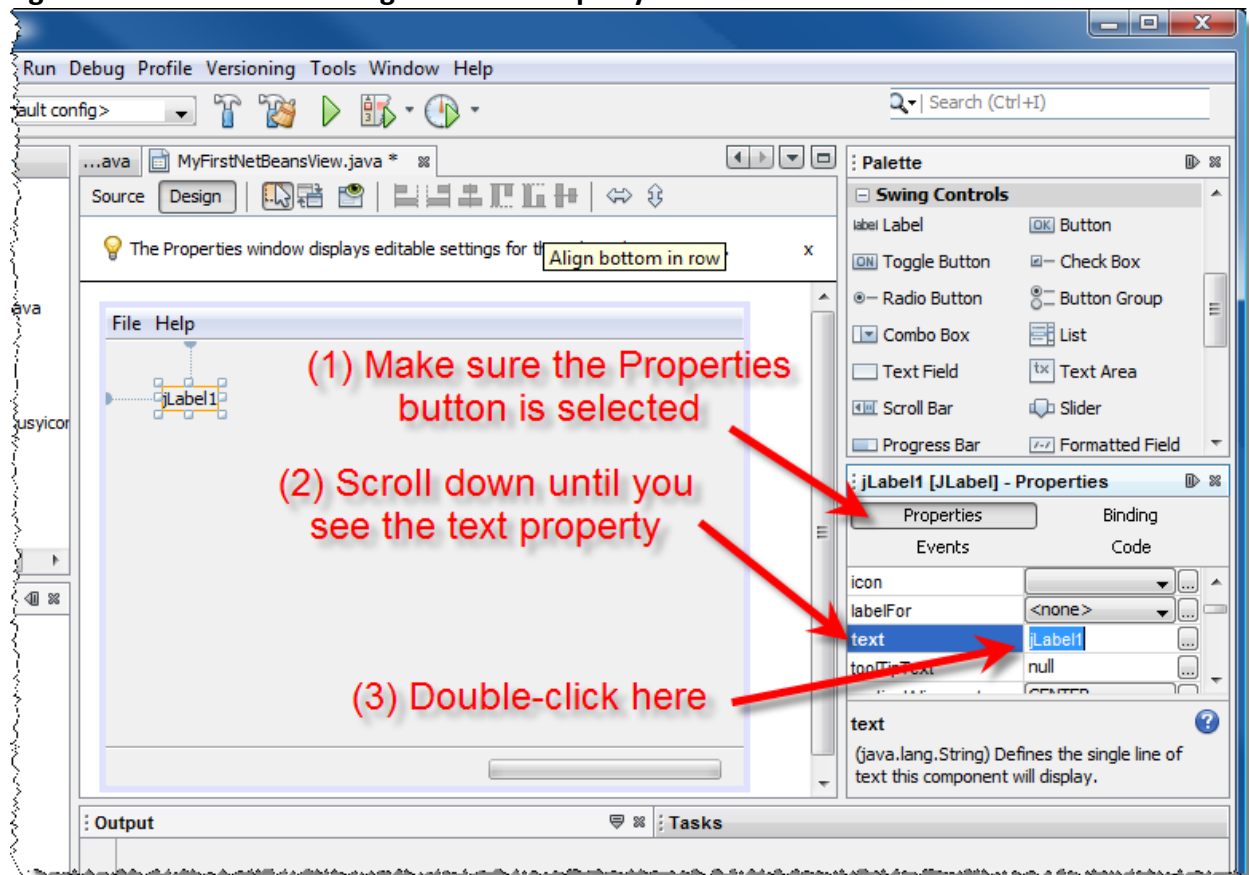
Figure 15-9 The Label's Text Changed
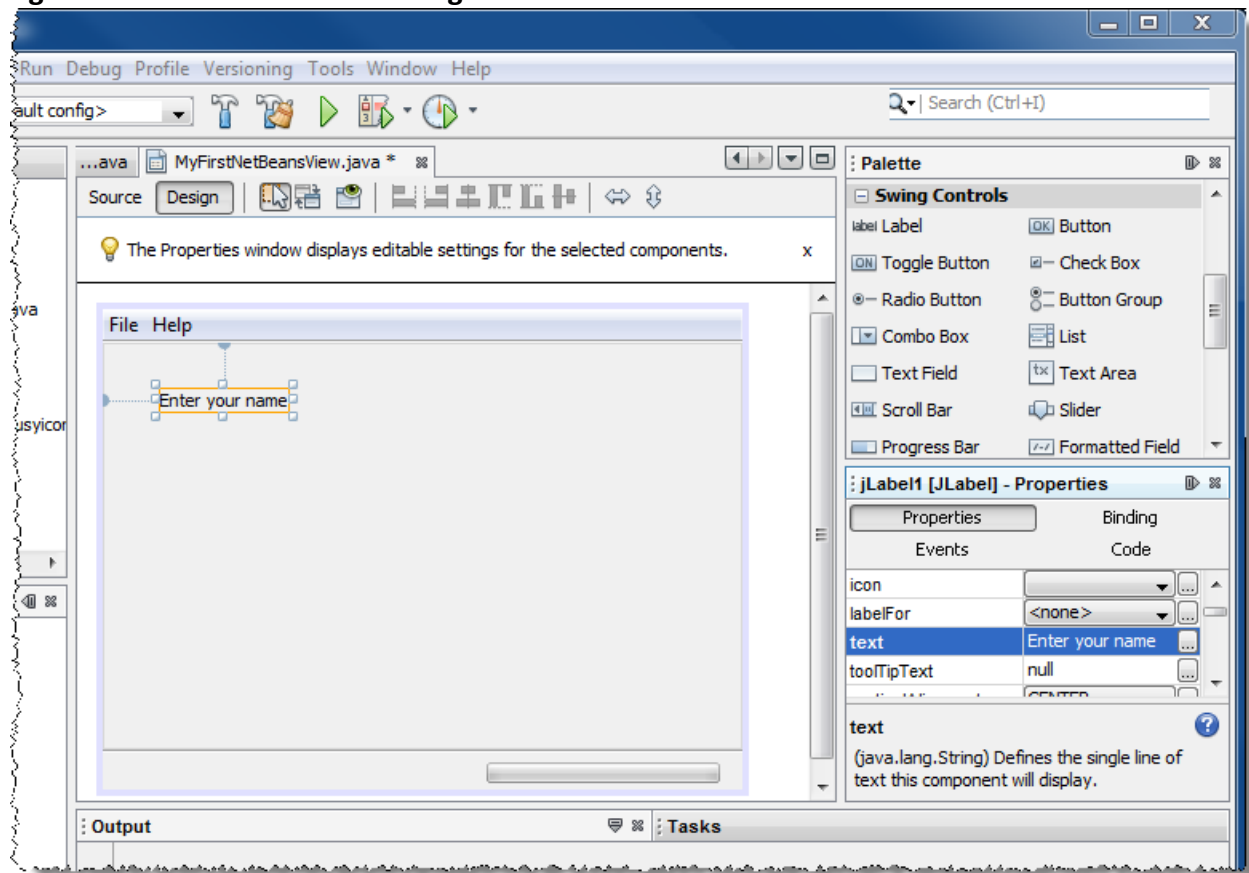
STEP 8: Now you will place a Text Field in the Design Area. Click and drag the Text Field control from the Palette Window into the Design Area. Drop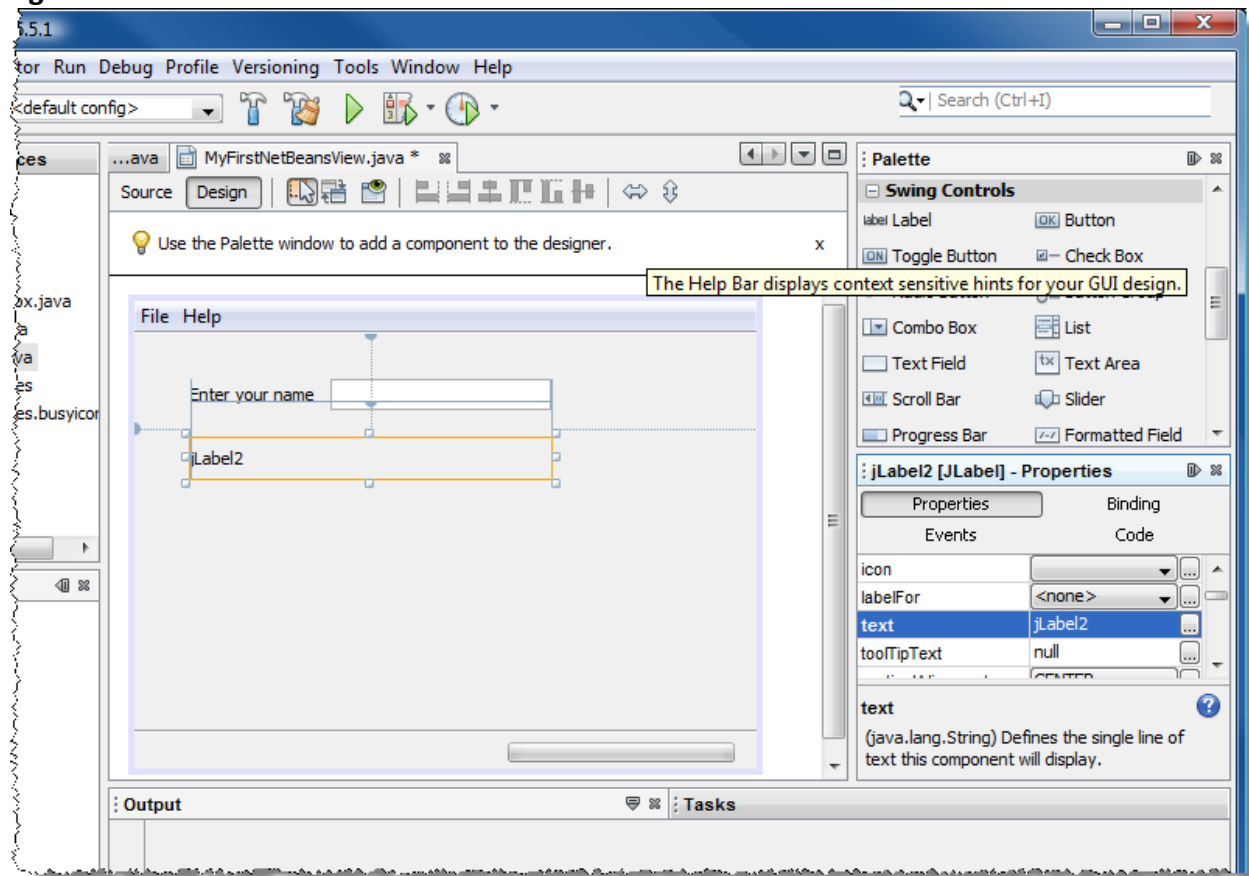 it in the approximate location shown in Figure 15-10. The name of the Text Field is jTextField1, and that is the text that is currently displayed in the Text Field. We don't want that text to be displayed in the Text Field when the application runs, so find the text property in the Property Window, double-click its value, press the Delete key, and then press Enter. This erases the text that is currently displayed in the Text Field.

The action that you just performed also causes the Text Field to shrink in size. In the Design Window, use the sizing handles to adjust the Text Field's size as shown in Figure 15-11.

Figure 15-10 A Text Field Placed in the Design Area
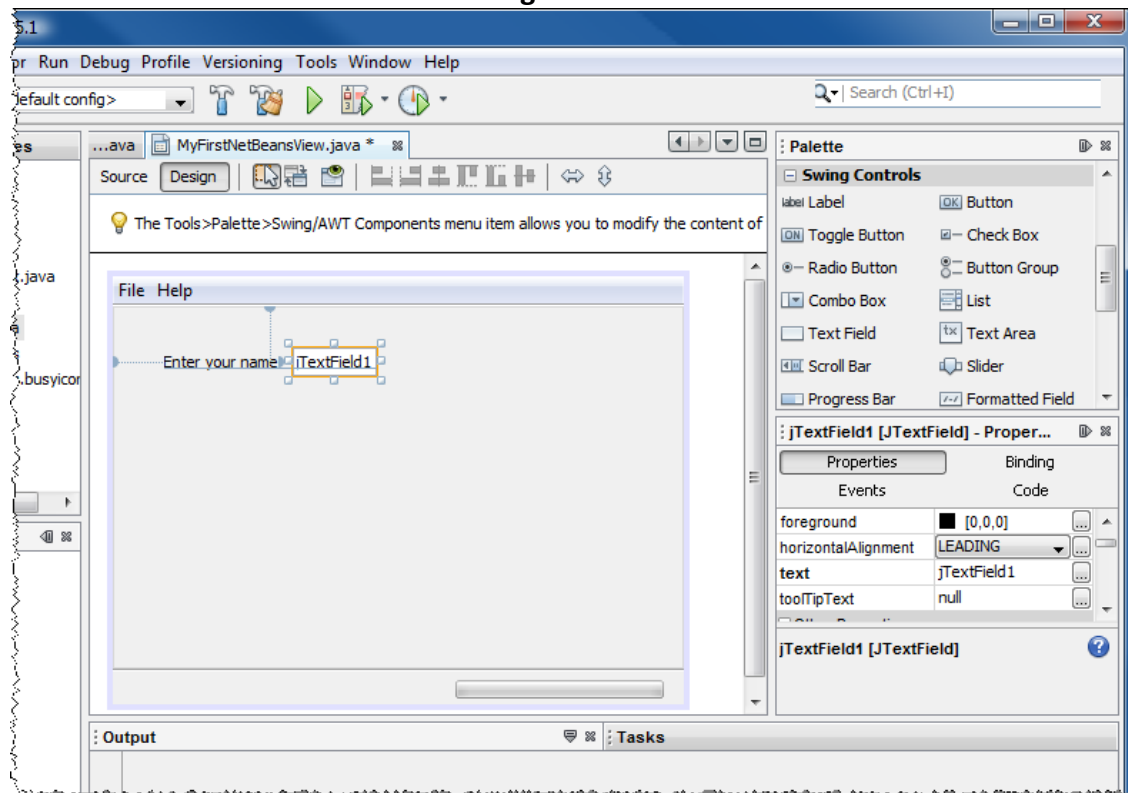
Figure 15-11 The Text Field Resized
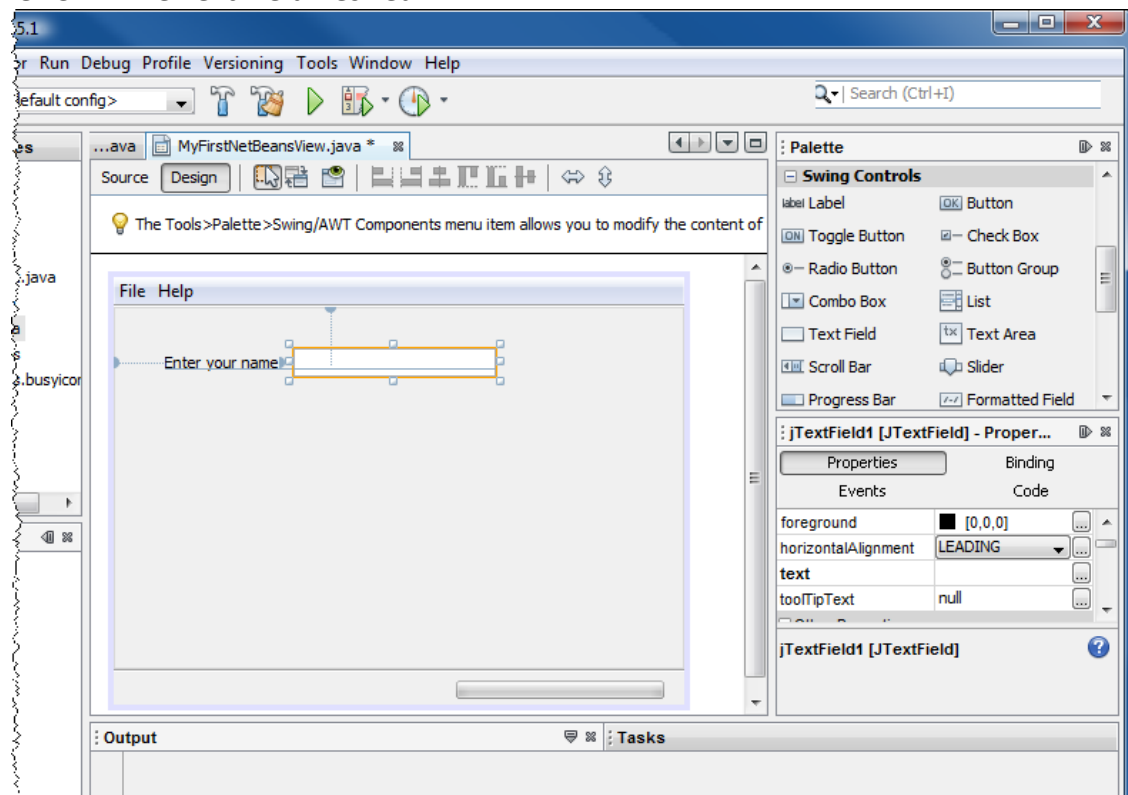
STEP 9: Now you will place another Label in the Design Area. Click and drag the Label control from the Palette Window into the Design Area. Drop it in the approximate location shown in Figure 15-12.

Figure 15-12 A Second Label Placed in the Design Area
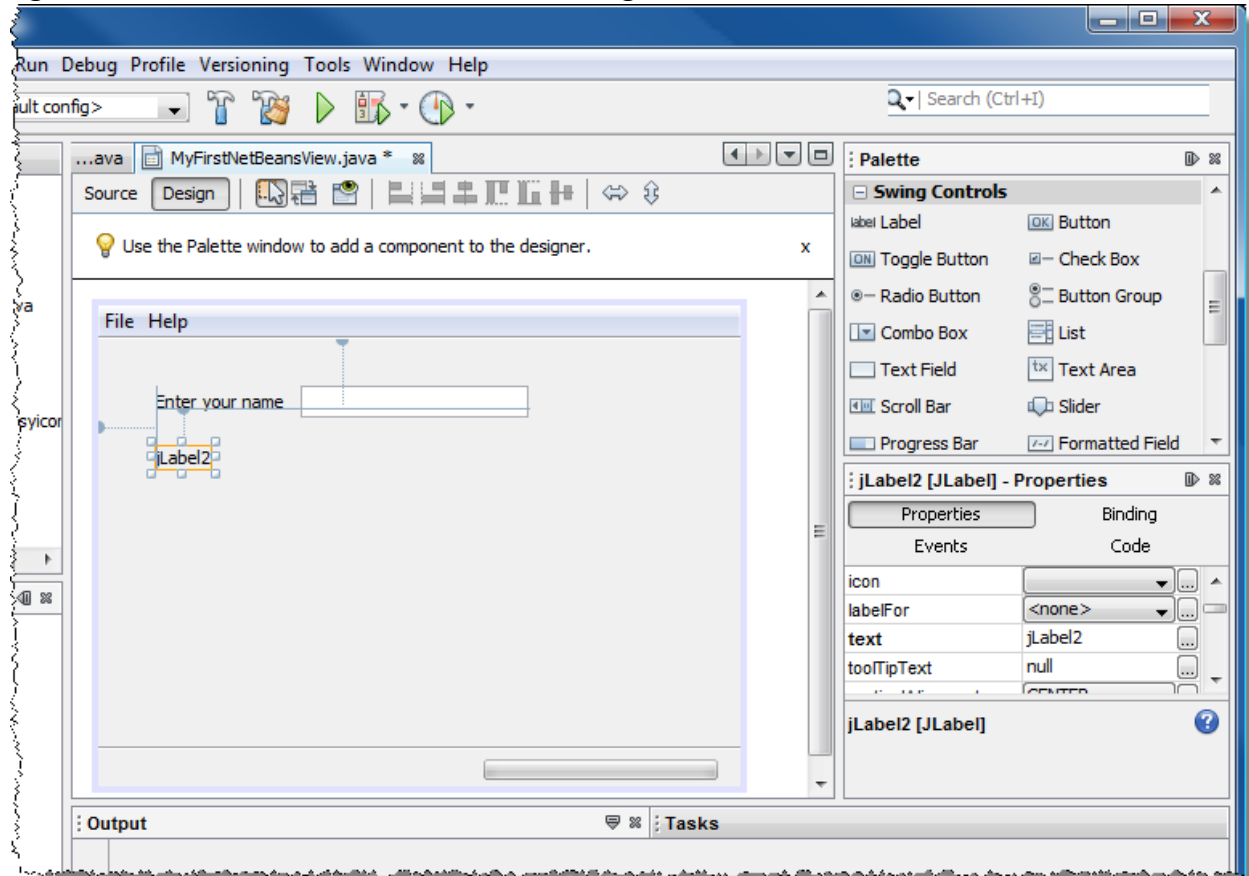
STEP 10: We want the new Label to be larger than the default size. Click and drag the Label's resizing handles so the Label appears similar to that shown in Figure 15-13. *Note: Enlarge the Label in both the horizontal and vertical directions.*

STEP 11: The name of the new Label is jLabel2, and that is also the text that is currently displayed in the Label. We don't want that text to be displayed in the Label when the application runs, so find the text property in the Property Window, double-click its value, press the Delete key, and then press Enter. This erases the text that is currently displayed in the Label. *Note: When you deselect the Label in the Design Area, it will disappear. That is because there is no text displayed by the Label. It is still there, though, and you can click it to see its sizing handles reappear.*

Figure 15-13 The Second Label Resized

STEP 12: Now you will place a Button the Design Area. Click and drag the Button control from the Palette Window into the Design Area. Drop it in the approximate location shown in Figure 15-14. The name of the Button is jButton1, and that is the text that is currently displayed on the face of the Button. We don't want that text to be displayed on the Button when the application runs, so find the text property in the Property Window, double-click its value, and type Click Me. The Button should now appear as shown in Figure 15-15.

Figure 15-14 A Button Placed in the Design Area
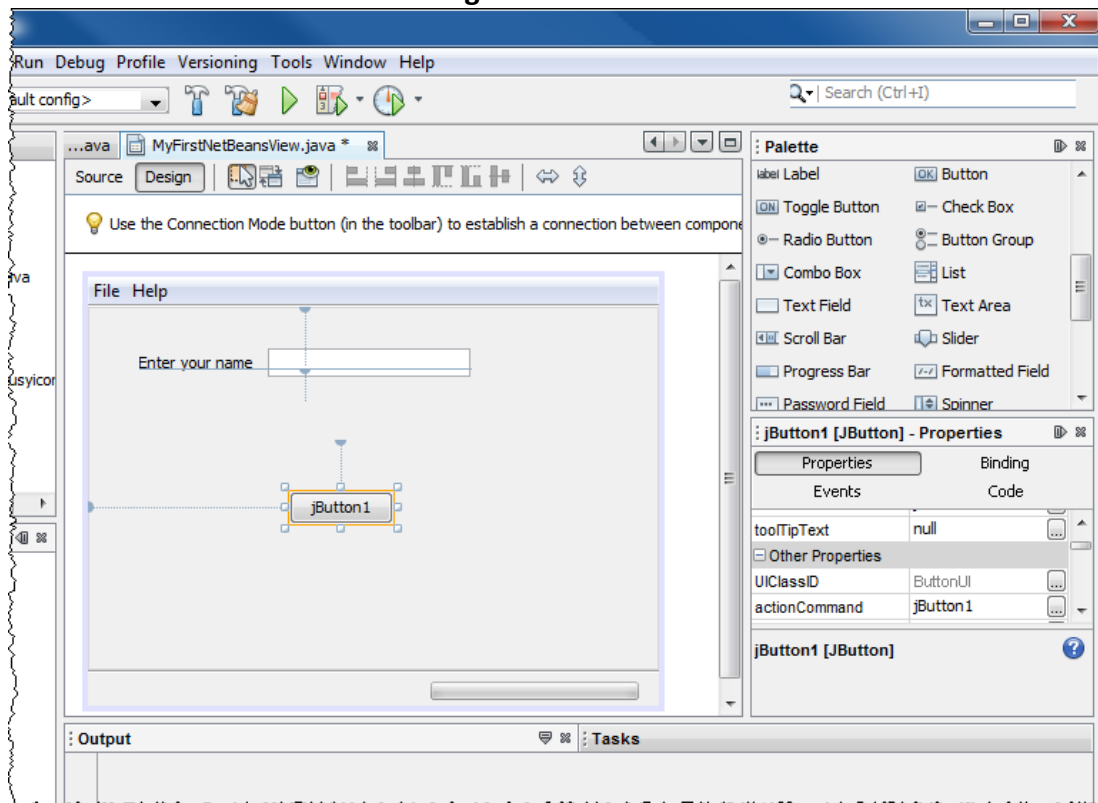
Figure 15-15 The Button's Text Changed
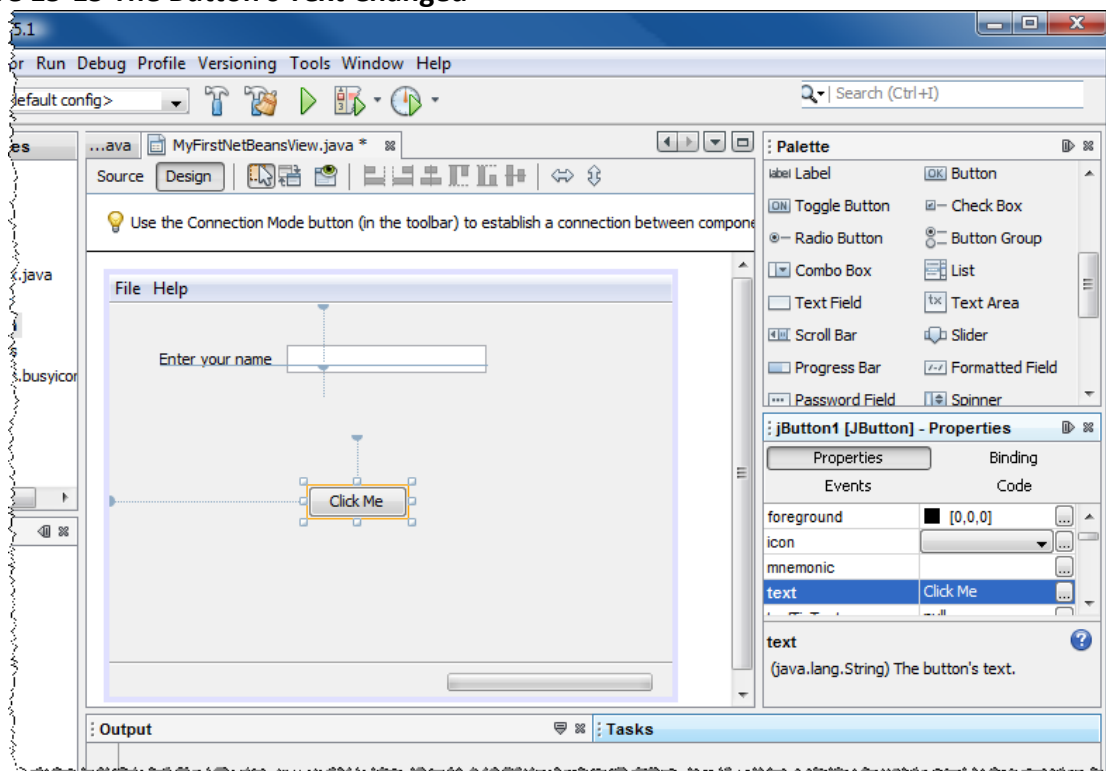
STEP 13: Now you will add an event handler to the button. The Event handler will perform the following actions when the user clicks the jButton1 button:

- It will retrieve the text that the user entered into the jTextField1 component. This value will be stored in the component's text property.
- It will create a string that reads "Hello " followed by the text that was retrieved from the jTextField1 component's text propery.
- It will display the string in the jLabel2 component.

Do the following steps:

- Make sure the jButton1 button is still selected.
- In the Properties Window, select the Events button as shown in Figure 15-16.
- Next to the actionPerformed event, click the … button as shown in Figure 15-17.
- A window will appear titled Handlers for actionPerformed. In this window, click the Add button.
- A dialog box titled Add Handler will appear. Type ButtonClick as the Handler Name, and click OK.
- Click OK again in the Handlers for actionPerformed window.
- You will now see the source code editor, as shown in Figure 15-18.
- In the area that reads `// TODO add your handing code here:`, type the following Java code:

  ```
  String str = "Hello " + jTextField1.getText();
  jLabel2.setText(str);
  ```

  Figure 15-19 shows the source code editor with this code added.

- Press Ctrl+Shift+S on the keyboard to save your project.
- Press F6 on the keyboard to compile and execute the application. When the application executes you will see the window shown in Figure 15-20. Type your name into the Text Field and then click the button. Clicking the button causes the event handling code to execute.
- The window should now appear as shown in Figure 15-21.
- Close the window.

That concludes the NetBeans tutorial! Click File, then Exit to close NetBeans.

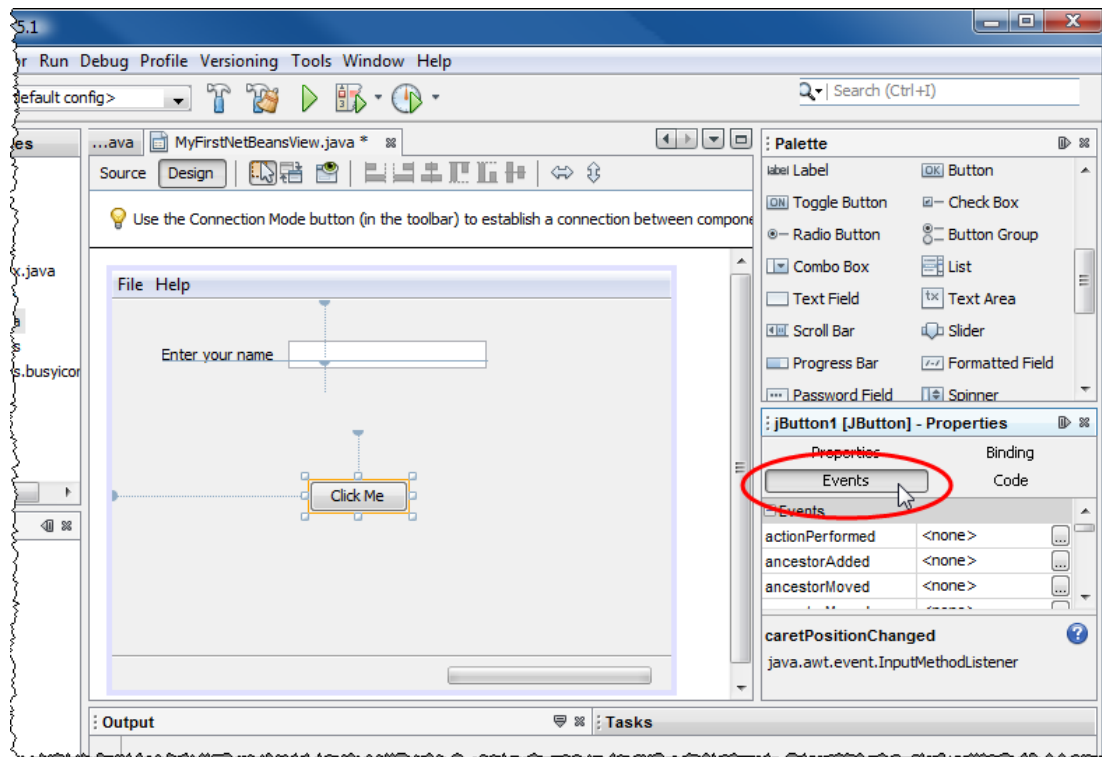**Figure 15-16 Select the Events Button in the Properties Window**
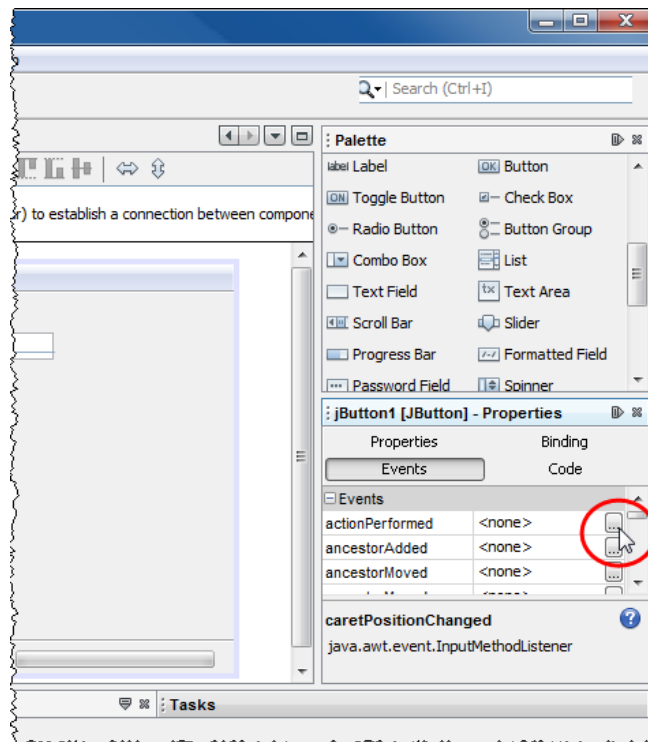

Figure 15-17 Click the … Button next to actionPerformed

Figure 15-18 Source Code Editor

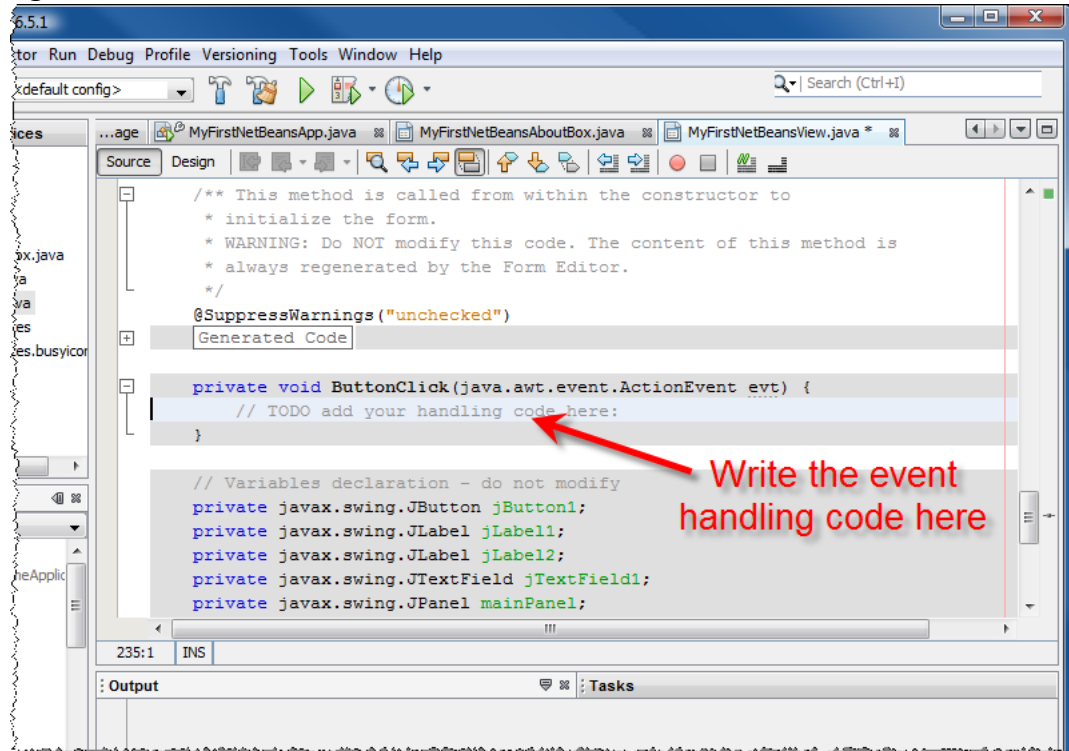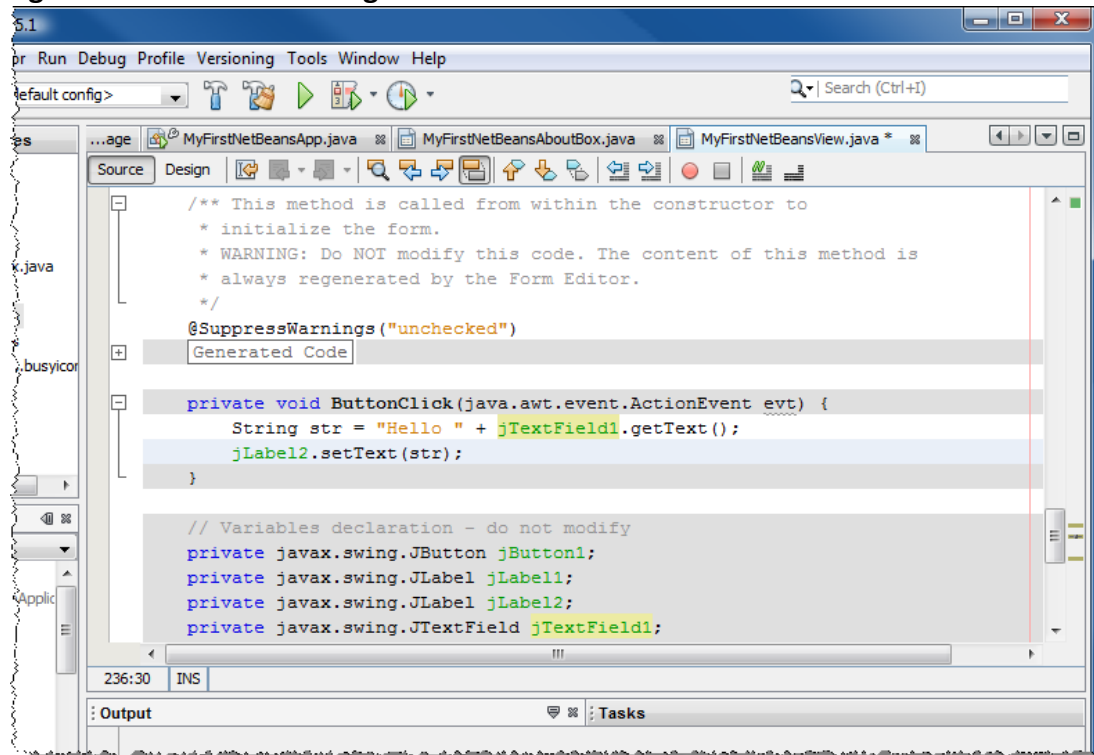**Figure 15-19 Event Handling Code Added**
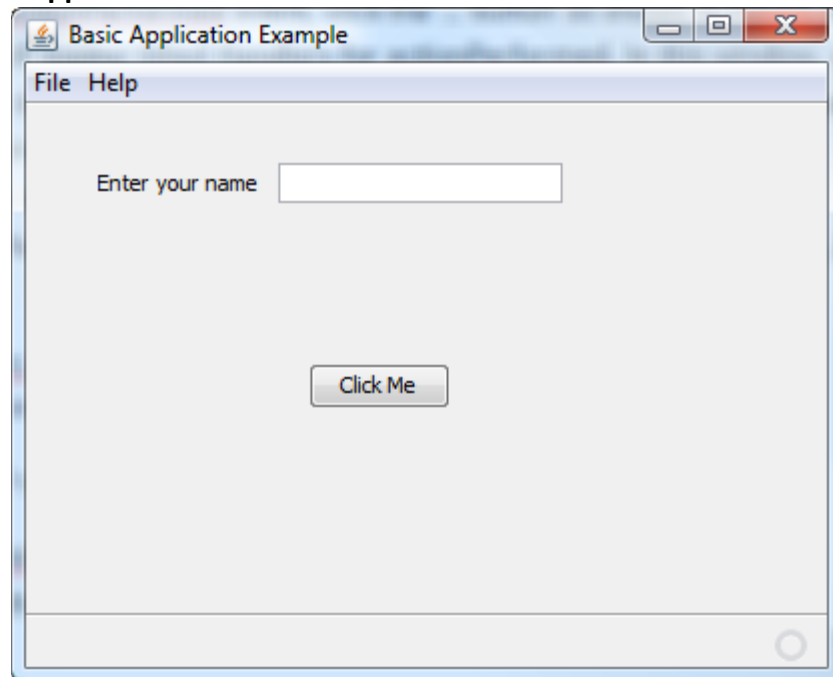
Figure 15-20 The Application's Window

Figure 15-21 After a Name is Entered and the Button Clicked