murach's
Java
programming

(Chapter 2)

Thanks for downloading this chapter from ***Murach's Java Programming***. We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its "how-to" headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our **web site**. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on Java development.

Thanks for your interest in our books!

MIKE MURACH & ASSOCIATES, INC.

1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • **www.murach.com**
Copyright © 2011 Mike Murach & Associates. All rights reserved.

What developers have said about previous editions of *Murach's Java*

"Finally there is a Java book for serious programmers doing real life business applications."

Donna Dean, IS Trainer, Chicago, Illinois

"I bought your Java book a week ago and I am already writing useful programs, not 'toys!'"

Richard Cooper, Programmer

"The absolute best teaching text for Java among the gazillions of self-hyped tutorials and skills books on the market today."

Posted at Amazon.com

"I love your Java book. It cuts right to the essential information, providing the perfect balance between too many details and too little information. Example apps are incredible. Keep up the good work."

Steve, Programmer, Denver, Colorado

"Terrific - Fantastic - Superlative! WELL worth several times the purchase price."

Posted at Amazon.com

"*Murach's Java* is now my go-to source for reference and learning and brushing up, well above and beyond the other books in my collection."

Jeff Salter, Sacramento Java Users Group (SacJUG)

"If I'd seen this book first, I would not have wasted money (and time) on 6 other books! This one is highly organized, clear, and very effective as a learning tool."

Posted at Amazon.com

"The style is clean, very user friendly, and simple. The narrative is totally accurate and focuses on important issues; it is not dumbed down. This book is a winner!"

Dr. Richard Wiener, Editor-in-Chief, *Journal of Object Technology*

2

Introduction to Java programming

Now that you know the basic skills for using NetBeans to work with Java projects, the quickest and best way to *learn* Java programming is to *do* Java programming. That's why this chapter shows you how to write complete Java programs that get input from a user, make calculations, and display output. When you finish this chapter, you should be able to write comparable programs of your own.

Basic coding skills

This chapter starts by introducing you to some basic coding skills. You'll use these skills for every Java program you develop.

How to code statements

The *statements* in a Java program direct the operation of the program. When you code a statement, you can start it anywhere in a coding line, you can continue it from one line to another, and you can code one or more spaces anywhere a single space is valid. In the first example in figure 2-1, the lines that aren't shaded are statements.

To end most statements, you use a semicolon. But when a statement requires a set of braces {}, it ends with the right brace. Then, the statements within the braces are referred to as a *block* of code. For example, the InvoiceApp class and the main method shown in this figure both contain a block of code.

To make a program easier to read, you should use indentation and spacing to align statements and blocks of code. This is illustrated by the program in this figure and by all of the programs and examples in this book.

How to code comments

Comments are used in Java programs to document what the program does and what specific blocks and lines of code do. Since the Java compiler ignores comments, you can include them anywhere in a program without affecting your code. In the first example in figure 2-1, the comments are shaded.

A *single-line comment* is typically used to describe one or more lines of code. This type of comment starts with two slashes (//) that tell the compiler to ignore all characters until the end of the current line. In the first example in this figure, you can see four single-line comments that are used to describe groups of statements. The other comment is coded after a statement. This type of comment is sometimes referred to as an *end-of-line comment*.

The second example in this figure shows how to code a *block comment*. This type of comment is typically used to document information that applies to a block of code. This information can include the author's name, program completion date, the purpose of the code, the files used by the code, and so on.

Although many programmers sprinkle their code with comments, that shouldn't be necessary if you write code that's easy to read and understand. Instead, you should use comments only to clarify code that's difficult to understand. In this figure, for example, an experienced Java programmer wouldn't need any of the single-line comments.

One problem with comments is that they may not accurately represent what the code does. This often happens when a programmer changes the code, but doesn't change the comments that go along with it. Then, it's even harder to understand the code because the comments are misleading. So if you change the code that you've written comments for, be sure to change the comments too.

An application consists of statements and comments

```java
import java.util.Scanner;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        // display a welcome message
        System.out.println("Welcome to the Invoice Total Calculator");
        System.out.println();   // print a blank line

        // get the input from the user
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter subtotal:    ");
        double subtotal = sc.nextDouble();

        // calculate the discount amount and total
        double discountPercent = .2;
        double discountAmount = subtotal * discountPercent;
        double invoiceTotal = subtotal - discountAmount;

        // format and display the result
        String message = "Discount percent: " + discountPercent + "\n"
                       + "Discount amount:   " + discountAmount + "\n"
                       + "Invoice total:     " + invoiceTotal + "\n";
        System.out.println(message);
    }
}
```

A block comment that could be coded at the start of a program

```java
/*
 * Author:   J. Murach
 * Purpose: This program uses the console to get a subtotal from the user,
 *          and it calculates the discount amount and total and displays them.
 */
```

Description

- Java *statements* direct the operations of a program, while *comments* are used to help document what the program does.

- You can start a statement at any point in a line and continue the statement from one line to the next. To make a program easier to read, you should use indentation and extra spaces to align statements and parts of statements.

- Most statements end with a semicolon. But when a statement requires a set of braces { }, the statement ends with the right brace. Then, the code within the braces can be referred to as a *block* of code.

- To code a *single-line comment*, type // followed by the comment. You can code a single-line comment on a line by itself or after a statement. A comment that's coded after a statement is sometimes called an *end-of-line comment*.

- To code a *block comment*, type /* at the start of the block and */ at the end. You can also code asterisks to identify the lines in the block, but that isn't necessary.

Figure 2-1   How to code statements and comments

How to create identifiers

As you code a Java program, you need to create and use *identifiers*. These are the names in the program that you define. In each program, for example, you need to create an identifier for the name of the program and for the variables that are used by the program.

Figure 2-2 shows you how to create identifiers. In brief, you must start each identifier with a letter, underscore, or dollar sign. After that first character, you can use any combination of letters, underscores, dollar signs, or digits.

Since Java is case-sensitive, you need to be careful when you create and use identifiers. If, for example, you define an identifier as CustomerAddress, you can't refer to it later as Customeraddress. That's a common coding error.

When you create an identifier, you should try to make the name both meaningful and easy to remember. To make a name meaningful, you should use as many characters as you need, so it's easy for other programmers to read and understand your code. For instance, netPrice is more meaningful than nPrice, and nPrice is more meaningful than np.

To make a name easy to remember, you should avoid abbreviations. If, for example, you use nwCst as an identifier, you may have difficulty remembering whether it was nCust, nwCust, or nwCst later on. If you code the name as newCustomer, though, you won't have any trouble remembering what it was. Yes, you type more characters when you create identifiers that are meaningful and easy to remember, but that will be more than justified by the time you'll save when you test, debug, and maintain the program.

For some common identifiers, though, programmers typically use just one or two lowercase letters. For instance, they often use the letters *i*, *j*, and *k* to identify counter variables. You'll see examples of this later in this chapter.

Notice that you can't create an identifier that is the same as one of the Java *keywords*. These 50 keywords are reserved by the Java language and are the basis for that language. To help you identify keywords in your code, Java IDEs display these keywords in a different color than the rest of the Java code. For example, NetBeans displays keywords in blue. As you progress through this book, you'll learn how to use almost all of these keywords.

Valid identifiers

```
InvoiceApp          $orderTotal     i
Invoice             _orderTotal     x
InvoiceApp2         input_string    TITLE
subtotal            _get_total      MONTHS_PER_YEAR
discountPercent     $_64_Valid
```

The rules for naming an identifier

- Start each identifier with a letter, underscore, or dollar sign. Use letters, dollar signs, underscores, or digits for subsequent characters.
- Use up to 255 characters.
- Don't use Java keywords.

Keywords

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

Description

- An *identifier* is any name that you create in a Java program. These can be the names of classes, methods, variables, and so on.
- A *keyword* is a word that's reserved by the Java language. As a result, you can't use keywords as identifiers.
- When you refer to an identifier, be sure to use the correct uppercase and lowercase letters because Java is a case-sensitive language.

Figure 2-2   How to create identifiers

How to declare a class and a main method

In the last chapter, you learned that if you use NetBeans to create a project, it can generate a *main class* with a *main method* for the project. Although you saw the code for four different classes and three different main methods in chapter 1, figure 2-3 now presents the syntax for declaring any *class* or main method. Even if you use NetBeans, you should be familiar with this syntax.

To code a class, you begin with a *class declaration*. In the syntax for declaring a class, the boldfaced words are Java keywords, and the words that aren't boldfaced represent code that the programmer supplies. The bar (|) in this syntax means that you have a choice between the two items that the bar separates. In this case, the bar means that you can start the declaration with the public keyword or the private keyword.

The public and private keywords are *access modifiers* that control the *scope* of a class. Usually, a class is declared public, which means that other classes can access it. Later in this book, you'll learn when and how to use private classes.

After the public keyword and the class keyword, you code the name of the class using the basic rules for creating an identifier. When you do, it's a common coding convention to start every word within a class name with a capital letter and to use letters and digits only. We also recommend that you use a noun or a noun that's preceded by one or more adjectives for your class names.

After the class name, the syntax summary shows a left brace, the statements that make up the class, and a right brace. It's a good coding practice, though, to type your ending brace right after you type the starting brace to prevent missing braces. When you use NetBeans, the ending brace is automatically added after you type the starting brace and then press the Enter key.

The two InvoiceApp classes in this figure show how a class works. Notice that the only difference between the two classes is where the opening braces for the class and the block of code within the class are placed. Although either technique is acceptable, we've chosen to use the first technique for this book whenever possible.

Within a class, you code one or more *methods*, which are pieces of code that perform the actions of the application (they're similar to *functions* in some programming languages). As you know, the main method is a special kind of method that's automatically executed when the class that contains it is run. All Java programs contain a main method that starts the program.

To code a main method, you begin by coding a *main method declaration* within the class declaration as shown in the two InvoiceApp classes in this figure. Although I won't describe this declaration, you should know that all main method declarations are coded exactly as shown. You'll learn more about the keywords used by this declaration later in this book.

To make the structure of the main method clear, it's indented and its starting and ending braces are aligned so it's easy to see where the method begins and ends. Then, between the braces, you can see the one statement that this main method performs. This statement displays a message to the user, and you'll learn more about it later in this chapter.

The syntax for declaring a class

```
public|private class ClassName
{
    statements
}
```

The syntax for declaring a main method

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

A public class named InvoiceApp that contains a main method

```
public class InvoiceApp                         // declare the class
{                                               // begin the class
    public static void main(String[] args)
    {
        System.out.println("Welcome to the Invoice Total Calculator");
    }
}                                               // end the class
```

The same class with different brace placement

```
public class InvoiceApp {                        // declare and begin the class
    public static void main(String[] args){
        System.out.println("Welcome to the Invoice Total Calculator");
    }
}                                               // end the class
```

The rules for naming a class

- Start the name with a capital letter.
- Use letters and digits only.
- Follow the other rules for naming an identifier.

Recommendations for naming a class

- Start every word within a class name with an initial cap.
- Each class name should be a noun or a noun that's preceded by one or more adjectives.

Description

- A Java application consists of one or more *classes* that start with a *class declaration*. You write the code for the class within the opening and closing braces of the declaration.
- The public and private keywords are *access modifiers* that control what parts of the program can use the class. Most classes are declared public, which means that the class can be used by all parts of the program.
- The file name for a class is the same as the class name with *java* as the extension.
- A *method* is a block of code that performs a task.
- Every Java application contains one *main method* that you can declare exactly as shown above. This is called the *main method declaration*.
- The statements between the braces in a main method declaration are run when the program is executed.

Figure 2-3   How to declare a class and a main method

How to work with numeric variables

In this topic, you'll learn how to work with numeric variables. This will introduce you to the use of variables, assignment statements, arithmetic expressions, and two of the eight primitive data types that are supported by Java. Then, you can learn all the details about working with the primitive data types in the next chapter.

How to declare and initialize variables

A *variable* is used to store a value that can change as a program executes. Before you can use a variable, you must *declare* its data type and name, and you must *assign* a value to it to *initialize* it. The easiest way to do that is shown in figure 2-4. Just code the data type, the variable name, the equals sign, and the value that you want to assign to the variable.

This figure also summarizes two of the eight Java *data types*. You can use the *int* data type to store *integers*, which are numbers that don't contain decimal places (whole numbers), and you can use the *double* data type to store numbers that contain decimal places. In the next chapter, you'll learn how to use the six other primitive data types, but these are the two that you'll probably use the most.

As you can see in the summary, the double data type can be used to store numbers with up to 16 *significant digits*. In case you aren't familiar with significant digits, they include any digit that identifies the precision of a number. That includes any non-zero digit as well as any zero digits between two non-zero digits and any zero digits at the end of a number with decimal places. For business applications, you'll rarely need to use numbers with more than 16 significant digits.

To illustrate the declaration of variables, the first example in this figure declares an int variable named scoreCounter with an initial value of 1. And the second example declares a double variable named unitPrice with an initial value of 14.95. When you assign values to double types, it's a good coding practice to include a decimal point, even if the initial value is a whole number. If, for example, you want to assign the number 29 to the variable, you should code the number as 29.0.

If you follow the naming recommendations in this figure as you name the variables, it will make your programs easier to read and understand. In particular, you should capitalize the first letter in each word of the variable name, except the first word, as in scoreCounter or unitPrice. This is commonly referred to as *camel notation*.

When you initialize a variable, you can assign a *literal* value like 1 or 14.95 to the variable as illustrated by the examples in this figure. However, you can also initialize a variable to the value of another variable as shown in the second example in this figure or to the value of an expression like the arithmetic expressions shown in the next figure.

Two of the eight primitive data types

Type	Description
`int`	Integers from -2,147,483,648 to 2,147,483,647.
`double`	Numbers with decimal places and up to 16 significant digits.

How to declare and initialize a variable in one statement

Syntax

```
type variableName = value;
```

Examples

```
int scoreCounter = 1;            // initialize an integer variable
double unitPrice = 14.95;        // initialize a double variable
```

How to code assignment statements

```
int quantity = 0;                // initialize an integer variable
int maxQuantity = 100;           // initialize another integer variable

// two assignment statements
quantity = 10;                   // quantity is now 10
quantity = maxQuantity;          // quantity is now 100
```

Description

- A *variable* stores a value that can change as a program executes.

- Java provides for eight *primitive data types* that you can use for storing values in memory. The two that you'll use the most are the int and double data types. In the next chapter, you'll learn how to use the other primitive data types.

- The *int* data type is used for storing *integers* (whole numbers). The *double* data type is used for storing numbers that can have one or more decimal places.

- Before you can use a variable, you must *declare* its data type and *assign* an initial value to the variable. It's common to *initialize* integer variables to 0 and double variables to 0.0.

- An *assignment statement* assigns a value to a variable. This value can be a literal value, another variable, or an expression like the arithmetic expressions that you'll learn how to code in the next figure. If a variable has already been declared, the assignment statement doesn't include the data type of the variable.

Naming recommendations for variables

- Start variable names with a lowercase letter and capitalize the first letter in all words after the first word.

- Each variable name should be a noun or a noun preceded by one or more adjectives.

- Try to use meaningful names that are easy to remember.

Figure 2-4   How to declare and initialize variables

How to code assignment statements

After you declare a variable, you can assign a new value to it. To do that, you code an *assignment statement*. In a simple assignment statement, you code the variable name, an equals sign, and a new value. The new value can be a literal value or the name of another variable as shown in figure 2-4. Or, the new value can be the result of an expression like the arithmetic expressions shown in figure 2-5.

How to code arithmetic expressions

To code simple *arithmetic expressions*, you can use the four *arithmetic operators* that are summarized in figure 2-5. As the first group of statements shows, these operators work the way you would expect them to with one exception. If you divide one integer into another integer, any decimal places are truncated. In contrast, if you divide a double into a double, the decimal places are included in the result.

When you code assignment statements, it's common to code the same variable on both sides of the equals sign. For example, you can add 1 to the value of a variable named counter with a statement like this:

```
counter = counter + 1;
```

In this case, if counter has a value of 5 when the statement starts, it will have a value of 6 when the statement finishes. This concept is illustrated by the second and third groups of statements.

What happens when you mix integer and double variables in the same arithmetic expression? The integers are *cast* (converted) to doubles so the decimal places can be included in the result. To retain the decimal places, though, the result variable must be a double. This is illustrated by the fourth group of statements.

Although it's not shown in this figure, you can also code expressions that contain two or more operators. When you do that, you need to be sure that the operations are done in the correct sequence. You'll learn more about that in the next chapter.

The basic operators that you can use in arithmetic expressions

Operator	Name	Description
+	Addition	Adds two operands.
-	Subtraction	Subtracts the right operand from the left operand.
*	Multiplication	Multiplies the right operand and the left operand.
/	Division	Divides the right operand into the left operand. If both operands are integers, then the result is an integer.

Statements that use simple arithmetic expressions

```
// integer arithmetic
int x = 14;
int y = 8;
int result1 = x + y;          // result1 = 22
int result2 = x - y;          // result2 = 6
int result3 = x * y;          // result3 = 112
int result4 = x / y;          // result4 = 1

// double arithmetic
double a = 8.5;
double b = 3.4;
double result5 = a + b;       // result5 = 11.9
double result6 = a - b;       // result6 = 5.1
double result7 = a * b;       // result7 = 28.9
double result8 = a / b;       // result8 = 2.5
```

Statements that increment a counter variable

```
int invoiceCount = 0;
invoiceCount = invoiceCount + 1;                   // invoiceCount = 1
invoiceCount = invoiceCount + 1;                   // invoiceCount = 2
```

Statements that add amounts to a total

```
double invoiceAmount1 = 150.25;
double invoiceAmount2 = 100.75;
double invoiceTotal = 0.0;
invoiceTotal = invoiceTotal + invoiceAmount1;   // invoiceTotal = 150.25
invoiceTotal = invoiceTotal + invoiceAmount2;   // invoiceTotal = 251.00
```

Statements that mix int and double variables

```
int result9 = invoiceTotal / invoiceCount       // result9 = 125
double result10 = invoiceTotal / invoiceCount   // result10 = 125.50
```

Description

- An *arithmetic expression* consists of one or more *operands* and *arithmetic operators*.

- When an expression mixes the use of int and double variables, Java automatically *casts* the int types to double types. To retain the decimal places, the variable that receives the result must be a double.

- In the next chapter, you'll learn how to code expressions that contain two or more operators.

Figure 2-5   How to code arithmetic expressions

How to work with string variables

In the topics that follow, you'll learn some basic skills for working with strings. For now, these skills should be all you need for many of the programs you develop. Keep in mind, though, that many programs require extensive string operations. That's why chapter 13 covers strings in more detail.

How to create a String object

A *string* can consist of any letters, numbers, and special characters. To declare a string variable, you use the syntax shown in figure 2-6. Although this is much like the syntax for declaring a numeric variable, a string is an object that's created from the String class when a string variable is declared. Then, the String object refers to string data. When you declare a string variable, you must capitalize the String keyword because it is the name of a class, not a primitive data type.

In the next topic and in chapter 7, you'll learn more about classes and objects. For now, though, all you need to know is that string variables work much like numeric variables, except that they store string data instead of numeric data.

When you declare a String object, you can assign a *string literal* to it by enclosing the characters within double quotes. You can also assign an *empty string* to it by coding a set of quotation marks with nothing between them. Finally, you can use the null keyword to assign a *null value* to a String object. That indicates that the value of the string is unknown.

How to join and append strings

If you want to *join*, or *concatenate*, two or more strings into one, you can use the + operator. For example, you can join a first name, a space, and a last name as shown in the second example in figure 2-6. Then, you can assign that string to a variable. Notice that when concatenating strings, you can use string variables or string literals.

You can also join a string with a primitive data type. This is illustrated in the third example in this figure. Here, a variable that's defined with the double data type is appended to a string. When you use this technique, Java automatically converts the double value to a string.

You can use the + and += operators to *append* a string to the end of a string that's stored in a string variable. If you use the + operator, you need to include the variable on both sides of the = operator. Otherwise, the assignment statement will replace the old value with the new value instead of appending the old value to the new value. Since the += operator provides a shorter and safer way to append strings, this operator is commonly used.

The syntax for declaring and initializing a string variable

```
String variableName = value;
```

Example 1: How to declare and initialize a string

```
String message1 = "Invalid data entry.";
String message2 = "";
String message3 = null;
```

Example 2: How to join strings

```
String firstName = "Bob";                    // firstName is Bob
String lastName = "Smith";                   // lastName is Smith
String name = firstName + " " + lastName;    // name is Bob Smith
```

Example 3: How to join a string and a number

```
double price = 14.95;
String priceString = "Price: " + price;
```

Example 4: How to append one string to another with the + operator

```
firstName = "Bob";                 // firstName is Bob
lastName = "Smith";                // lastName is Smith
name = firstName + " ";            // name is Bob followed by a space
name = name + lastName;            // name is Bob Smith
```

Example 5: How to append one string to another with the += operator

```
firstName = "Bob";                 // firstName is Bob
lastName = "Smith";                // lastName is Smith
name = firstName + " ";            // name is Bob followed by a space
name += lastName;                  // name is Bob Smith
```

Description

- A *string* can consist of any characters in the character set including letters, numbers, and special characters like *, &, and #.
- In Java, a string is actually a String object that's created from the String class that's part of the Java *API* (*Application Programming Interface*). The API provides all the classes that are included as part of the JDK.
- To specify the value of a string, you can enclose text in double quotation marks. This is known as a *string literal*.
- To assign a *null value* to a string, you can use the null keyword. This means that the value of the string is unknown.
- To assign an *empty string* to a String object, you can code a set of quotation marks with nothing between them. This means that the string doesn't contain any characters.
- To *join* (or *concatenate*) a string with another string or a data type, use a plus sign. Whenever possible, Java will automatically convert the data type so it can be used as part of the string.
- When you *append* one string to another, you add one string to the end of another. To do that, you can use assignment statements.
- The += operator is a shortcut for appending a string expression to a string variable.

Figure 2-6   How to create and use strings

How to include special characters in strings

Figure 2-7 shows how to include certain types of special characters within a string. In particular, this figure shows how to include backslashes, quotation marks, and control characters such as new lines, tabs, and returns in a string. To do that, you can use the *escape sequences* shown in this figure.

As you can see, each escape sequence starts with a backslash. The backslash tells the compiler that the character that follows should be treated as a special character and not interpreted as a literal value. If you code a backslash followed by the letter *n*, for example, the compiler will include a new line character in the string. You can see how this works in the first example in this figure. If you omitted the backslash, of course, the compiler would just include the letter *n* in the string value. The escape sequences for the tab and return characters work similarly, as you can see in the second example.

To code a string literal, you enclose it in double quotes. If you want to include a double quote within a string literal, then, you must use an escape sequence. This is illustrated in the third example. Here, the \" escape sequence is used to include two double quotes within the string literal.

Finally, you need to use an escape sequence if you want to include a backslash in a string literal. To do that, you code two backslashes as shown in the fourth example. If you code a single backslash, the compiler will treat the next character as a special character. That will cause a compiler error if the character isn't a valid special character. And if the character is a valid special character, the results won't be what you want.

Common escape sequences

Sequence	Character
\n	New line
\t	Tab
\r	Return
\"	Quotation mark
\\	Backslash

Example 1: New line

String
```
"Code: JSPS\nPrice: $49.50"
```

Result
```
Code: JSPS
Price: $49.50
```

Example 2: Tabs and returns

String
```
"Joe\tSmith\rKate\tLewis"
```

Result
```
Joe     Smith
Kate    Lewis
```

Example 3: Quotation marks

String
```
"Type \"x\" to exit"
```

Result
```
Type "x" to exit
```

Example 4: Backslash

String
```
"C:\\java\\files"
```

Result
```
C:\java\files
```

Description

- Within a string, you can use *escape sequences* to include certain types of special characters.

Figure 2-7   How to include special characters in strings

How to use Java classes, objects, and methods

So far, you've learned how to create String objects from the String class in the Java API. As you develop Java applications, though, you need to use dozens of different Java classes and objects. To do that, you need to know how to create objects from Java classes, how to call Java methods, and how to import Java classes.

How to create objects and call methods

To use a Java *class*, you usually start by creating an *object* from the class. As the syntax in figure 2-8 shows, you do that by coding the Java class name, the name that you want to use for the object, an equals sign, the new keyword, and the Java class name again followed by a set of parentheses. Within the parentheses, you code any *arguments* that are required by the *constructor* of the object that's defined in the class.

In the examples, the first statement shows how to create a Scanner object named sc. The constructor for this object requires just one argument (System.in), which represents console input. In contrast, the second statement creates a Date object named now that represents the current date, but its constructor doesn't require any arguments. As you go through this book, you'll learn how to create objects with constructors that require two or more arguments, and you'll see that a single class can provide more than one constructor for creating objects.

When you create an object, you can think of the class as the template for the object. That's why the object can be called an *instance* of the class, and the process of creating the object can be called *instantiation*. Whenever necessary, you can create more than one object or instance from the class. For instance, you often use several String objects in a single program.

Once you've created an object from a class, you can use the *methods* of the class. To *call* one of these methods, you code the object name, a dot (period), and the method name followed by a set of parentheses. Within the parentheses, you code the arguments that are required by the method.

In the examples, the first statement calls the nextDouble method of the Scanner object named sc to get data from the console. The second statement calls the toString method of the Date object named now to convert the date and time that's stored in the object to a string. Neither one of these methods requires an argument, but you'll soon see some that do.

Besides methods that you can call from an object, some classes provide *static methods* that can be called directly from the class. To do that, you substitute the class name for the object name as illustrated by the third set of examples. Here, the first statement calls the toString method of the Double class, and the second statement calls the parseDouble method of the Double class. Both of these methods require one argument.

How to create an object from a class

Syntax

```
ClassName objectName = new ClassName(arguments);
```

Examples

```
Scanner sc = new Scanner(System.in);   // creates a Scanner object named sc
Date now = new Date();                  // creates a Date object named now
```

How to call a method from an object

Syntax

```
objectName.methodName(arguments)
```

Examples

```
double subtotal = sc.nextDouble();     // get a double entry from the console
String currentDate = now.toString();   // convert the date to a string
```

How to call a static method from a class

Syntax

```
ClassName.methodName(arguments)
```

Examples

```
String sPrice = Double.toString(price);        // convert a double to a string
double total = Double.parseDouble(userEntry);  // convert a string to a double
```

Description

- When you create an *object* from a Java class, you are creating an *instance* of the *class*. Then, you can use the *methods* of the class by *calling* them from the object.

- Some Java classes contain *static methods*. These methods can be called directly from the class without creating an object.

- When you create an object from a class, the *constructor* may require one or more *arguments*. These arguments must have the required data types, and they must be coded in the correct sequence separated by commas.

- When you call a method from an object or a class, the method may require one or more arguments. Here again, these arguments must have the required data types and they must be coded in the correct sequence separated by commas.

- Although you can use the syntax shown in this figure to create a String object, the syntax in figure 2-6 is the preferred way to do that. Once a String object is created, though, you call its methods from the object as shown above.

- In this book, you'll learn how to use dozens of the Java classes and methods. You will also learn how to create your own classes and methods.

Figure 2-8   How to create objects and call methods

Incidentally, you can also use the syntax shown in this figure to create a String object. However, the preferred way to create a String object is to use the syntax shown in figure 2-6. Once a String object is created, though, you use the syntax in this figure to call one of its methods. You'll see examples of this later in this chapter.

In the pages and chapters that follow, you'll learn how to use dozens of classes and methods. For now, though, you just need to focus on the syntax for creating an object from a class, for calling a method from an object, and for calling a static method from a class. Once you understand that, you're ready to learn how to import the Java classes you need for your programs.

How to import Java classes

In the API for the Java SE, groups of related classes are organized into *packages*. In figure 2-9, you can see a list of some of the commonly used packages. Since the java.lang package contains the classes that are used in almost every Java program (such as the String class), this package is automatically made available to all programs.

To use a class from a package other than java.lang, though, you'll typically include an import statement for that class at the beginning of the program. If you don't, you'll still be able to use the class, but you'll have to qualify it with the name of the package that contains it each time you refer to it. Since that can lead to a lot of unnecessary typing, we recommend that you always code an import statement for the classes you use.

When you code an import statement, you can import a single class by specifying the class name, or you can import all of the classes in the package by typing an asterisk (*) in place of the class name. The first two statements in this figure, for example, import a single class, while the next two import all of the classes in a package. Although it requires less code to import all of the classes in a package at once, importing one class at a time clearly identifies the classes you're using.

As this figure shows, Java provides two different technologies for building a graphical user interface (GUI) that contains text boxes, command buttons, combo boxes, and so on. The older technology, known as the *Abstract Window Toolkit* (*AWT*), was used with versions 1.0 and 1.1 of Java. Its classes are stored in the java.awt package. Since version 1.2 of Java, though, a new technology known as *Swing* has been available. The Swing classes are stored in the javax.swing package. In general, many of the newer package names begin with javax instead of java. Here, the x indicates that these packages can be considered extensions to the original Java API.

In addition to the packages provided by the Java API, you can get packages from third party sources, either as open-source code or by purchasing them. For more information, check the Java web site. You can also create packages that contain classes that you've written. You'll learn how to do that in chapter 10.

Common packages

Package name	Description
`java.lang`	Provides classes fundamental to Java, including classes that work with primitive data types, strings, and math functions.
`java.text`	Provides classes to handle text, dates, and numbers.
`java.util`	Provides various utility classes including those for working with collections.
`java.io`	Provides classes to read data from files and to write data to files.
`java.sql`	Provides classes to read data from databases and to write data to databases.
`java.applet`	An older package that provides classes to create an applet.
`java.awt`	An older package called the *Abstract Window Toolkit* (AWT) that provides classes to create graphical user interfaces.
`java.awt.event`	A package that provides classes necessary to handle events.
`javax.swing`	A newer package called *Swing* that provides classes to create graphical user interfaces and applets.

The syntax of the import statement

```
import packagename.ClassName;
    or
import packagename.*;
```

Examples

```
import java.text.NumberFormat;
import java.util.Scanner;
import java.util.*;
import javax.swing.*;
```

How to use the Scanner class to create an object

With an import statement

```
Scanner sc = new Scanner(System.in);
```

Without an import statement

```
java.util.Scanner sc = new java.util.Scanner(System.in);
```

Description

- The API for the Java SE provides a large library of classes that are organized into *packages*.

- All classes stored in the java.lang package are automatically available to all Java programs.

- To use classes that aren't in the java.lang package, you can code an import statement as shown above. To import one class from a package, specify the package name followed by the class name. To import all classes in a package, specify the package name followed by an asterisk (*).

- If you don't code an import statement for a class, you must qualify the class name with the name of the package that contains it each time you refer to the class.

Figure 2-9   How to import Java classes

How to use the API documentation
to research Java classes

If you refer back to figure 2-2, you can see that the Java language consists of just 50 keywords that you can master with relative ease. What's difficult about using Java, though, is mastering the hundreds of classes and methods that your applications will require. To do that, you frequently need to study the API documentation that comes with Java, and that is one of the most time-consuming aspects of Java programming.
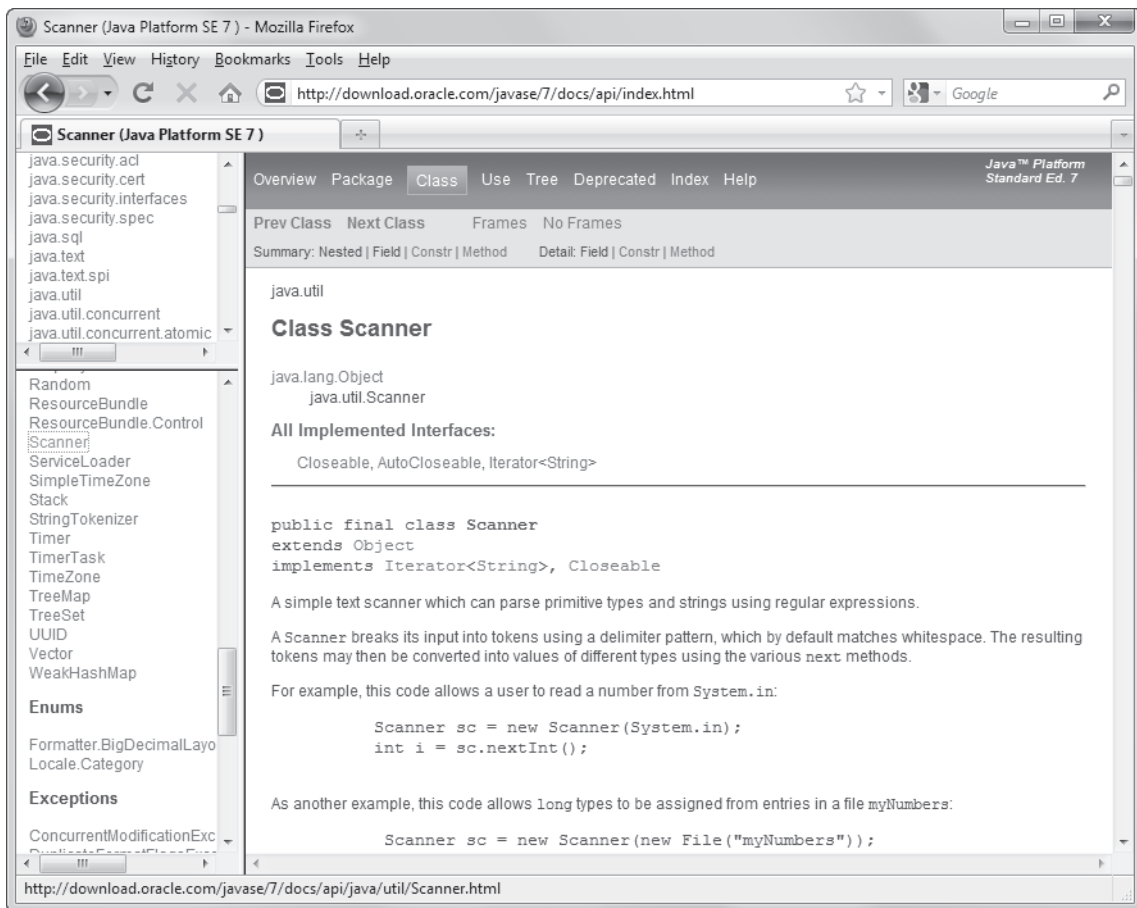
Figure 2-10 summarizes some of the basic techniques for navigating through the API documentation. Here, you can see the start of the documentation for the Scanner class, which goes on for many pages. To get there, you click on the package name in the upper left frame and then on the class name in the lower left frame.

If you scroll through the documentation for this class, you'll get an idea of the scale of the documentation that you're dealing with. After a few pages of descriptive information, you come to a summary of the eight constructors for the class. After that, you come to a summary of the dozens of methods that the class offers. That in turn is followed by more detail about the constructors, and then by more detail about the methods.

At this point in your development, this is far more information than you can handle. That's why one of the goals of this book is to introduce you to the dozens of classes and methods that you'll use in most of the applications that you develop. Once you've learned those, the API documentation will make more sense to you, and you'll be able to use that documentation to research classes and methods that aren't presented in this book. To get you started with the use of objects and methods, figure 2-12 will show you how to use the Scanner class.

It's never too early to start using the documentation, though. So by all means use the documentation to get more information about the methods that are presented in this book and to research the other methods that are offered by the classes that are presented in this book. After you learn how to use the Scanner class, for example, take some time to do some research on that class. You'll get a chance to do that in exercise 2-4.

The documentation for the Scanner class

Description

- The Java SE API contains thousands of classes and methods that can help you do most of the tasks that your applications require.

- You can use a browser to view the Java SE API on the Internet by going to this address:

  http://download.oracle.com/javase/7/docs/api/index.html

- You can select the name of the package in the top left frame to display information about the package and the classes it contains. Then, you can select a class in the lower left frame to display the documentation for that class in the right frame.

- Once you display the documentation for a class, you can scroll through it or click on a hyperlink to get more information.

- To make it easier to access the API documentation, you should bookmark the index page. Then, you can easily redisplay this page whenever you need it.

Figure 2-10   How to use the API documentation to research Java classes

How to use the console for input and output

Most of the applications that you write will require some type of user interaction. In particular, most applications will get input from the user and display output to the user. Ever since version 1.5 of Java, the easiest way to get input is to use the new Scanner class to retrieve data from the console. And the easiest way to display output is to print it to the console.

How to use the System.out object to print output to the console

To print output to the *console*, you can use the println and print methods of the System.out object as shown in figure 2-11. Here, System.out refers to an instance of the PrintStream class. Because this object is created automatically by Java, you don't have to include code that creates it in your program.

Both the println and print methods accept a string argument that specifies the data to be printed. The only difference between the two is that the println method starts a new line after it displays the data, and the print method doesn't.

If you study the examples in this figure, you shouldn't have any trouble using these methods. For instance, the first statement in the first example uses the println method to print the words "Welcome to the Invoice Total Calculator" to the console. The second statement prints the string "Total: " followed by the value of the total variable (which is automatically converted to a string by this join). The third statement prints the value of the variable named message to the console. And the fourth statement prints a blank line since no argument is coded.

Because the print method doesn't automatically start a new line, you can use it to print several data arguments on the same line. For instance, the three statements in the second example use the print method to print "Total: ", followed by the total variable, followed by a new line character. Of course, you can achieve the same result with a single line of code like this:

```
System.out.println("Total: " + total);
```

This figure also shows an application that uses the println method to print seven lines to the console. In the main method of this application, the first four statements set the values for four variables. Then, the next seven statements print a welcome message, a blank line, the values for the four variables, and another blank line.

When you work with console applications, you should know that the appearance of the console may differ slightly depending on the operating system. Even if the console looks a little different, however, it should work the same.

Two methods of the System.out object

Method	Description
`println(data)`	Prints the data argument followed by a new line character to the console.
`print(data)`	Prints the data to the console without starting a new line.

Example 1: The println method

```
System.out.println("Welcome to the Invoice Total Calculator");
System.out.println("Total: " + total);
System.out.println(message);
System.out.println();            // print a blank line
```

Example 2: The print method

```
System.out.print("Total: ");
System.out.print(total);
System.out.print("\n");
```

Example 3: An application that prints data to the console

```
public class InvoiceApp
{
    public static void main(String[] args)
    {
        // set and calculate the numeric values
        double subtotal = 100;         // set subtotal to 100
        double discountPercent = .2;   // set discountPercent to 20%
        double discountAmount = subtotal * discountPercent;
        double invoiceTotal = subtotal - discountAmount;

        // print the data to the console
        System.out.println("Welcome to the Invoice Total Calculator");
        System.out.println();
        System.out.println("Subtotal:         " + subtotal);
        System.out.println("Discount percent: " + discountPercent);
        System.out.println("Discount amount:  " + discountAmount);
        System.out.println("Total:            " + invoiceTotal);
        System.out.println();
    }
}
```

The console output

```
Welcome to the Invoice Total Calculator

Subtotal:         100.0
Discount percent: 0.2
Discount amount:  20.0
Total:            80.0
```

Description

- Although the appearance of a console may differ from one system to another, you can always use the print and println methods to print data to the console.

Figure 2-11   How to use the System.out object to print output to the console

How to use the Scanner class to read input from the console

Figure 2-12 shows how you can use the Scanner class to read input from the console. To start, you create a Scanner object by using a statement like the one in this figure. Here, sc is the name of the Scanner object that is created and System.in represents console input, which is the keyboard. Like the System.out object, the System.in object is created by Java automatically. Because of that, you can use this object with a Scanner object whenever you want to get console input.

Once you've created a Scanner object, you can use the next methods to read data from the console. The method you use depends on the type of data you need to read. To read string data, for example, you use the next method. To read integer data, you use the nextInt method. To read double data, you use the nextDouble method. And to read all of the data on a line, you use the nextLine method.

The examples in this figure illustrate how these methods work. Here, the first statement gets a string and assigns it to a string variable named name. The second statement gets an integer and assigns it to an int variable named count. The third statement gets a double and assigns it to a double variable named subtotal. And the fourth statement reads any remaining characters on the line.

Each entry that a user makes is called a *token*, and a user can enter more than one token before pressing the Enter key. To do that, the user separates the entries by one or more space, tab, or return characters. This is called *whitespace*. Then, each next method gets the next token that has been entered. If, for example, you press the Enter key (a return character), type 100, press the Tab key, type 20, and press the Enter key again, the first token is 100 and the second one is 20.

If you want to get string data that includes whitespace, you can use the nextLine method. If, for example, the user enters "New York" and presses the Enter key, you can use the nextLine method to get the entire entry as a single string.

If the user doesn't enter the type of data that the next method is looking for, an error occurs and the program ends. In Java, an error like this is also known as an *exception*. If, for example, the user enters a double value but the nextInt method is used to get it, an exception occurs. In chapter 5, you'll learn how to prevent this type of error.

Although this figure only shows methods for working with String objects and int and double types, the Scanner class includes methods for working with most of the other data types that you'll learn about in the next chapter. It also includes methods that let you check what type of data the user entered. As you'll see in chapter 5, you can use these methods to avoid exceptions by checking the data type before you issue the next method.

The Scanner class

```
java.util.Scanner
```

How to create a Scanner object

```
Scanner sc = new Scanner(System.in);
```

Common methods of a Scanner object

Method	Description
`next()`	Returns the next token stored in the scanner as a String object.
`nextInt()`	Returns the next token stored in the scanner as an int value.
`nextDouble()`	Returns the next token stored in the scanner as a double value.
`nextLine()`	Returns any remaining input on the current line as a String object and advances the scanner to the next line.

How to use the methods of a Scanner object

```
String name = sc.next();
int count = sc.nextInt();
double subtotal = sc.nextDouble();
String cityName = sc.nextLine();
```

Description

- To create a Scanner object that gets input from the *console*, specify System.in in the parentheses.

- When one of the next methods of the Scanner class is run, the application waits for the user to enter data with the keyboard. To complete the entry, the user presses the Enter key.

- Each entry that a user makes is called a *token*. A user can enter two or more tokens by separating them with *whitespace*, which consists of one or more spaces, tab characters, or return characters.

- The entries end when the user presses the Enter key. Then, the first next, nextInt, or nextDouble method gets the first token; the second next, nextInt, or nextDouble method gets the second token; and so on. In contrast, the nextLine method gets all of the input or remaining input on the current line.

- If the user doesn't enter the type of data that the next method expects, an error occurs and the program ends. In Java, this type of error is called an *exception*. You'll learn more about this in chapter 5.

- Since the Scanner class is in the java.util package, you'll want to include an import statement whenever you use this class.

Note

- The Scanner class was introduced in version 1.5 of the JDK.

Figure 2-12   How to use the Scanner class to read input from the console

Examples that get input from the console

Figure 2-13 presents two examples that get input from the console. The first example starts by creating a Scanner object. Then, it uses the print method of the System.out object to prompt the user for three values, and it uses the next methods of the Scanner object to read those values from the console. Because the first value should be a string, the next method is used to read this value. Because the second value should be a double, the nextDouble method is used to read it. And because the third value should be an integer, the nextInt method is used to read it.

After all three values are read, a calculation is performed using the int and double values. Then, the data is formatted and the println method is used to display the data on the console. You can see the results of this code in this figure.

Unlike the first example, which reads one value per line, the second example reads three values in a single line. Here, the first statement uses the print method to prompt the user to enter three integer values. Then, the next three statements use the nextInt method to read those three values. This works because a Scanner object uses whitespace (spaces, tabs, or returns) to separate the data that's entered at the console into tokens.

Example 1: Code that gets three values from the user

```
// create a Scanner object
Scanner sc = new Scanner(System.in);

// read a string
System.out.print("Enter product code: ");
String productCode = sc.next();

// read a double value
System.out.print("Enter price: ");
double price = sc.nextDouble();

// read an int value
System.out.print("Enter quantity: ");
int quantity = sc.nextInt();

// perform a calculation and display the result
double total = price * quantity;
System.out.println();
System.out.println(quantity + " " + productCode
    + " @ " + price + " = " + total);
System.out.println();
```

The console after the program finishes

```
Enter product code: cshp
Enter price: 49.50
Enter quantity: 2

2 cshp @ 49.5 = 99.0
```

Example 2: Code that reads three values from one line

```
// read three int values
System.out.print("Enter three integer values: ");
int i1 = sc.nextInt();
int i2 = sc.nextInt();
int i3 = sc.nextInt();

// calculate the average and display the result
int total = i1 + i2 + i3;
int avg = total / 3;
System.out.println("Average: " + avg);
System.out.println();
```

The console after the program finishes

```
Enter three integer values: 99 88 92
Average: 93
```

Figure 2-13   Examples that get input from the console

How to code simple control statements

As you write programs, you need to determine when certain operations should be done and how long repetitive operations should continue. To do that, you code *control statements* like the if/else and while statements. This topic will get you started with the use of these statements, but first you need to learn how to write expressions that compare numeric and string variables.

How to compare numeric variables

Figure 2-14 shows how to code *Boolean expressions* that use the six *relational operators* to compare int and double data types. This type of expression evaluates to either true or false based on the result of the comparison, and the operands in the expression can be either variables or literals.

For instance, the first expression in the first set of examples is true if the value of the variable named discountPercent is equal to the literal value 2.3. The second expression is true if the value of subtotal is not equal to zero. And the sixth expression is true if the value of the variable named quantity is less than or equal to the value of the variable named reorderPoint.

Although you shouldn't have any trouble coding simple expressions like these, you must remember to code two equals signs instead of one for the equality comparison. That's because a single equals sign is used for assignment statements. As a result, if you try to code a Boolean expression with a single equals sign, your code won't compile.

When you compare numeric values, you usually compare values of the same data type. However, if you compare different types of numeric values, Java will automatically cast the less precise numeric type to the more precise type. For example, if you compare an int type to a double type, the int type will be cast to the double type before the comparison is made.

How to compare string variables

Because a string is an object, not a primitive data type, you can't use the relational operators to compare strings. Instead, you must use the equals or equalsIgnoreCase methods of the String class that are summarized in figure 2-14. As you can see, both of these methods require an argument that provides the String object or literal that you want to compare with the current String object.

In the examples, the first expression is true if the value in the string named userEntry equals the literal value "Y". In contrast, the second expression uses the equalsIgnoreCase method, so it's true whether the value in userEntry is "Y" or "y". Then, the third expression shows how you can use the not operator (!) to reverse the value of a Boolean expression that compares two strings. Here, the expression will evaluate to true if the lastName variable is *not* equal to "Jones". The fourth expression is true if the string variable named code equals the string variable named productCode.

Relational operators

Operator	Name	Description
==	Equality	Returns a true value if both operands are equal.
!=	Inequality	Returns a true value if the left and right operands are not equal.
>	Greater Than	Returns a true value if the left operand is greater than the right operand.
<	Less Than	Returns a true value if the left operand is less than the right operand.
>=	Greater Than Or Equal	Returns a true value if the left operand is greater than or equal to the right operand.
<=	Less Than Or Equal	Returns a true value if the left operand is less than or equal to the right operand.

Examples of conditional expressions

```
discountPercent == 2.3     // equal to a numeric literal
subtotal != 0              // not equal to a numeric literal
years > 0                  // greater than a numeric literal
i < months                 // less than a numeric variable
subtotal >= 500            // greater than or equal to a numeric literal
quantity <= reorderPoint   // less than or equal to a numeric variable
```

Two methods of the String class

Method	Description
equals(String)	Compares the value of the String object with a String argument and returns a true value if they are equal. This method makes a case-sensitive comparison.
equalsIgnoreCase(String)	Works like the equals method but is not case-sensitive.

Examples

```
userEntry.equals("Y")                  // equal to a string literal
userEntry.equalsIgnoreCase("Y")        // equal to a string literal
(!lastName.equals("Jones"))            // not equal to a string literal
code.equalsIgnoreCase(productCode)     // equal to another string variable
```

Description

- You can use the *relational operators* to compare two numeric operands and return a *Boolean value* that is either true or false.

- To compare two numeric operands for equality, make sure to use two equals signs. If you only use one equals sign, you'll code an assignment statement, and your code won't compile.

- If you compare an int with a double, Java will cast the int to a double.

- To test two strings for equality, you must call one of the methods of the String object. If you use the equality operator, you will get unpredictable results (more about this in chapter 4).

Figure 2-14   How to compare numeric and string variables

How to code if/else statements

Figure 2-15 shows how to use the *if/else statement* (or just *if statement*) to control the logic of your applications. This statement is the Java implementation of a control structure known as the *selection structure* because it lets you select different actions based on the results of a Boolean expression.

As you can see in the syntax summary, you can code this statement with just an if clause, you can code it with one or more else if clauses, and you can code it with a final else clause. In any syntax summary, the ellipsis (…) means that the preceding element (in this case the else if clause) can be repeated as many times as it is needed. And the brackets [ ] mean that the element is optional.

When an if statement is executed, Java begins by evaluating the Boolean expression in the if clause. If it's true, the statements within this clause are executed and the rest of the if/else statement is skipped. If it's false, Java evaluates the first else if clause (if there is one). Then, if its Boolean expression is true, the statements within this else if clause are executed, and the rest of the if/else statement is skipped. Otherwise, Java evaluates the next else if clause.

This continues with any remaining else if clauses. Finally, if none of the clauses contains a Boolean expression that evaluates to true, Java executes the statements in the else clause (if there is one). However, if none of the Boolean expressions are true and there is no else clause, Java doesn't execute any statements.

If a clause only contains one statement, you don't need to enclose that statement in braces. This is illustrated by the first statement in the first example in this figure. However, if you want to code two or more statements within a clause, you need to code the statements in braces. The braces identify the block of statements that is executed for the clause.

If you declare a variable within a block, that variable is available only to the other statements in the block. This can be referred to as *block scope*. As a result, if you need to access a variable outside of the block, you should declare it before the if statement. You'll see this illustrated by the program at the end of this chapter.

The syntax of the if/else statement

```
if (booleanExpression) {statements}
[else if (booleanExpression) {statements}] ...
[else {statements}]
```

Example 1: If statements without else if or else clauses

With a single statement

```
if (subtotal >= 100)
    discountPercent = .2;
```

With a block of statements

```
if (subtotal >= 100)
{
    discountPercent = .2;
    status = "Bulk rate";
}
```

Example 2: An if statement with an else clause

```
if (subtotal >= 100)
    discountPercent = .2;
else
    discountPercent = .1;
```

Example 3: An if statement with else if and else clauses

```
if (customerType.equals("T"))
    discountPercent = .4;
else if (customerType.equals("C"))
    discountPercent = .2;
else if (subtotal >= 100)
    discountPercent = .2;
else
    discountPercent = .1;
```

Description

- An *if/else statement*, or just *if statement*, always contains an if clause. In addition, it can contain one or more else if clauses, and a final else clause.

- If a clause requires just one statement, you don't have to enclose the statement in braces.

- If a clause requires more than one statement, you enclose the block of statements in braces.

- Any variables that are declared within a block have *block scope* so they can only be used within that block.

Figure 2-15   How to code if/else statements

How to code while statements

Figure 2-16 shows how to code a *while statement*. This is one way that Java implements a control structure know as the *iteration structure* because it lets you repeat a block of statements. As you will see in chapter 4, though, Java also offers other implementations of this structure.

When a while statement is executed, the program repeats the statements in the block of code within the braces *while* the expression in the statement is true. In other words, the statement ends when the expression becomes false. If the expression is false when the statement starts, the statements in the block of code are never executed.

Because a while statement loops through the statements in the block as many times as needed, the code within a while statement is often referred to as a *while loop*. Here again, any variables that are defined within the block have block scope, which means that they can't be accessed outside the block.

The first example in this figure shows how to code a loop that executes a block of statements while a variable named choice is equal to either "y" or "Y". In this case, the statements within the block get input from the console, process it, and display output. This is a common way to control the execution of a program, and you'll see this illustrated in detail in the next figure.

The second example shows how to code a loop that adds the numbers 1 through 4 and stores the result in a variable named sum. Here, a *counter variable* (or just *counter*) named i is initialized to 1 and the sum variable is initialized to zero before the loop starts. Then, each time through the loop, the value of i is added to sum and one is added to i. Because the value of i is 1 the first time through the loop, for example, 1 is added to sum so its value becomes 1. The second time through the loop, 2 is added to sum so its value becomes 3. The third time through the loop, 3 is added to sum so its value becomes 6. And the fourth time through the loop, 4 is added to sum so its value becomes 10. When the value of i becomes 5, though, the expression in the while statement is no longer true and the loop ends. The use of a counter like this is a common coding practice, and single letters like *i, j,* and *k* are commonly used as the names of counters.

When you code loops, you must be careful to avoid *infinite loops*. If, for example, you forget to code a statement that increments the counter variable in the second example, the loop will never end because the counter will never get to 5. Then, you have to cancel the application so you can debug your code. In NetBeans, you can do that by clicking on the Stop button that's available from the Output window when a console application is running.

The syntax of the while loop

```
while (booleanExpression)
{
    statements
}
```

Example 1: A loop that continues while choice is "y" or "Y"

```
String choice = "y";
while (choice.equalsIgnoreCase("y"))
{
    // get the invoice subtotal from the user
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter subtotal:   ");
    double subtotal = sc.nextDouble();

    // the code that processes the user's entry goes here

    // see if the user wants to continue
    System.out.print("Continue? (y/n): ");
    choice = sc.next();
    System.out.println();
}
```

Example 2: A loop that calculates the sum of the numbers 1 through 4

```
int i = 1;
int sum = 0;
while (i < 5)
{
    sum = sum + i;
    i = i + 1;
}
```

Description

- A *while statement* executes the block of statements within its braces as long as the Boolean expression is true. When the expression becomes false, the while statement skips its block of statements so the program continues with the next statement in sequence.
- The statements within a while statement can be referred to as a *while loop*.
- Any variables that are declared in the block of a while statement have block scope.
- If the Boolean expression in a while statement never becomes false, the statement never ends. Then, the program goes into an *infinite loop*. In NetBeans, you can cancel an infinite loop by clicking on the Stop button in the Output window.

Figure 2-16   How to code while loops

Two illustrative applications

You have now learned enough about Java to write simple applications of your own. To show you how you can do that, this chapter ends by presenting two illustrative applications.

The Invoice application

In chapter 1, you saw how the console looks when displayed by NetBeans. Now, this figure shows the console in a platform-neutral format that's easy to read. This is the format that will be used to display console output for the rest of this book.

Figure 2-17 shows the console and code for an Invoice application. Although this application is simple, it gets input from the user, performs calculations that use this input, and displays the results of the calculations. It continues until the user enters anything other than "Y" or "y" in response to the Continue prompt.

The Invoice application starts by displaying a welcome message at the console. Then, it creates a Scanner object named sc that will be used in the while loop of the program. Although this object could be created within the while loop, that would mean that the object would be recreated each time through the loop, and that would be inefficient.

Before the while statement is executed, a String object named choice is initialized to "y". Then, the loop starts by getting a double value from the user and storing it in a variable named subtotal. After that, the loop uses an if/else statement to calculate the discount amount based on the value of subtotal. If, for example, subtotal is greater than or equal to 200, the discount amount is .2 times the subtotal (a 20% discount). If that condition isn't true but subtotal is greater than or equal to 100, the discount is .1 times the subtotal (a 10% discount). Otherwise, the discount amount is zero. When the if/else statement is finished, an assignment statement calculates the invoice total by subtracting discountAmount from subtotal.

At that point, the program displays the discount percent, discount amount, and invoice total on the console. Then, it displays a message that asks the user if he or she wants to continue. If the user enters "y" or "Y", the loop is repeated. Otherwise, the program ends.

Although this application illustrates most of what you've learned in this chapter, it has a couple of shortcomings. First, the numeric values that are displayed should be formatted with two decimal places since these are currency values. In the next chapter, you'll learn how to do that type of formatting.

Second, an exception will occur and the program will end prematurely if the user doesn't enter one valid double value for the subtotal each time through the loop. This is a serious problem that isn't acceptable in a professional program, and you'll learn how to prevent problems like this in chapter 5.

In the meantime, if you're new to programming, you can learn a lot by writing simple programs like the Invoice program. That will give you a chance to become comfortable with the coding for input, calculations, output, if/else statements, and while statements.

The console input and output for a test run

```
Welcome to the Invoice Total Calculator

Enter subtotal:    150
Discount percent: 0.1
Discount amount:  15.0
Invoice total:    135.0


Continue? (y/n):
```

The code for the application

```java
import java.util.Scanner;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        System.out.println("Welcome to the Invoice Total Calculator");
        System.out.println();  // print a blank line

        Scanner sc = new Scanner(System.in);

        // perform invoice calculations until choice isn't equal to "y" or "Y"
        String choice = "y";
        while (choice.equalsIgnoreCase("y"))
        {
            // get the invoice subtotal from the user
            System.out.print("Enter subtotal:    ");
            double subtotal = sc.nextDouble();

            // calculate the discount amount and total
            double discountPercent = 0.0;
            if (subtotal >= 200)
                discountPercent = .2;
            else if (subtotal >= 100)
                discountPercent = .1;
            else
                discountPercent = 0.0;

            double discountAmount = subtotal * discountPercent;
            double total = subtotal - discountAmount;

            // display the discount amount and total
            String message = "Discount percent: " + discountPercent + "\n"
                          + "Discount amount:  " + discountAmount + "\n"
                          + "Invoice total:    " + total + "\n";
            System.out.println(message);

            // see if the user wants to continue
            System.out.print("Continue? (y/n): ");
            choice = sc.next();
            System.out.println();
        }
    }
}
```

Figure 2-17   The Invoice application

The Test Score application

Figure 2-18 presents another Java application that will give you more ideas for how you can apply what you've learned so far. If you look at the console input and output for this application, you can see that it lets the user enter one or more test scores. To end the application, the user enters a value of 999. Then, the application displays the number of test scores that were entered, the total of the scores, and the average of the scores.

If you look at the code for this application, you can see that it starts by displaying the instructions for using the application. Then, it declares and initializes three variables, and it creates a Scanner object that will be used to get console input.

The while loop in this program continues until the user enters a test score that's greater than 100. To start, this loop gets the next test score. Then, if that test score is less than or equal to 100, the program adds one to scoreCount, which keeps track of the number of scores, and adds the test score to scoreTotal, which accumulates the total of the scores. The if statement that does this is needed, because you don't want to increase scoreCount and scoreTotal if the user enters 999 to end the program. When the loop ends, the program calculates the average score and displays the score count, total, and average.

To include decimal places in the score average, this program declares scoreTotal and averageScore as a double data types. Declaring scoreTotal as a double type causes the score average to be calculated with decimal places. Declaring the averageScore variable as a double type allows it to store those decimal places.

To allow statements outside of the while loop to access the scoreTotal and scoreCount variables, this program declares these variables before the while loop. If these variables were declared inside the while loop, they would only be available within that block of code and couldn't be accessed by the statements that are executed after the while loop. In addition, the logic of the program wouldn't work because these variables would be reinitialized each time through the loop.

Here again, this program has some obvious shortcomings that will be addressed in later chapters. First, the data isn't formatted properly, but you'll learn how to fix that in the next chapter. Second, an exception will occur and the program will end prematurely if the user enters invalid data, but you'll learn how to fix that in chapter 5.

The console input and output for a test run

```
Please enter test scores that range from 0 to 100.
To end the program enter 999.

Enter score: 90
Enter score: 80
Enter score: 75
Enter score: 999

Score count:    3
Score total:    245.0
Average score: 81.66666666666667
```

The code for the application

```java
import java.util.Scanner;

public class TestScoreApp
{
    public static void main(String[] args)
    {
        System.out.println(
            "Please enter test scores that range from 0 to 100.");
        System.out.println("To end the program enter 999.");
        System.out.println();  // print a blank line

        // initialize variables and create a Scanner object
        double scoreTotal = 0.0;
        int scoreCount = 0;
        int testScore = 0;
        Scanner sc = new Scanner(System.in);

        // get a series of test scores from the user
        while (testScore <= 100)
        {
            // get the input from the user
            System.out.print("Enter score: ");
            testScore = sc.nextInt();

            // accumulate score count and score total
            if (testScore <= 100)
            {
                scoreCount = scoreCount + 1;
                scoreTotal = scoreTotal + testScore;
            }
        }

        // display the score count, score total, and average score
        double averageScore = scoreTotal / scoreCount;
        String message = "\n"
                    + "Score count:    " + scoreCount + "\n"
                    + "Score total:    " + scoreTotal + "\n"
                    + "Average score: " + averageScore + "\n";
        System.out.println(message);
    }
}
```

Figure 2-18   The Test Score application

How to test and debug an application

In chapter 1, you were introduced to errors that are detected by NetBeans as you enter code, called *syntax errors*. Because syntax errors keep an application from being compiled, they are also commonly referred to as *compile-time errors*. Once you've fixed the syntax errors, you're ready to test and debug the application as described in this topic. Then, in the next two chapters, you'll learn several more debugging techniques. And when you do the exercises, you'll get lots of practice testing and debugging.

How to test an application

When you *test* an application, you run it to make sure the application works correctly. As you test, you should try every possible combination of valid and invalid data to be certain that the application works correctly under every set of conditions. Remember that the goal of testing is to find errors, or *bugs*, not to show that an application works correctly.

As you test, you will encounter two types of bugs. The first type of bug causes a *runtime error* also known as a *runtime exception*. A runtime error causes the application to end prematurely, which programmers often refer to as "crashing" or "blowing up." In this case, an error message like the one in the first example in figure 2-19 is displayed, and this message shows the line number of the statement that was being executed when the error occurred.

The second type of bug produces inaccurate results when an application runs. These bugs occur due to *logic errors* in the source code. For instance, the second example in this figure shows output for the Test Score application. Here, the final totals were displayed and the application ended before any input was entered. This type of bug can be more difficult to find and correct than a runtime error.
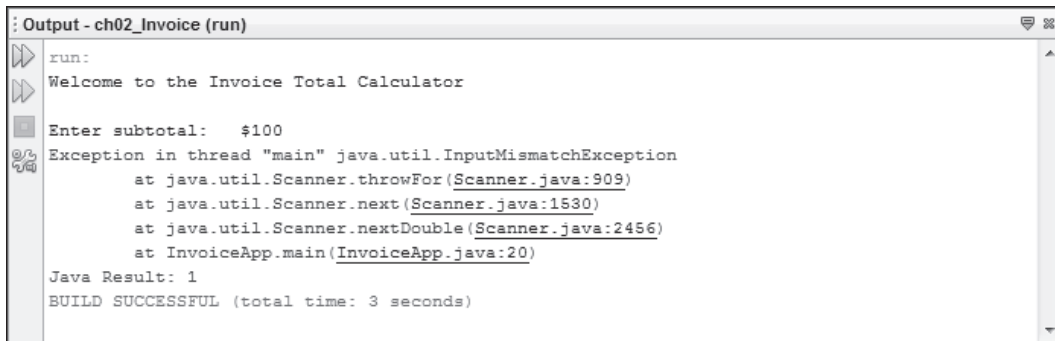
How to debug an application

When you *debug* a program, you find the cause of the bugs, fix them, recompile, and test again. As you progress through this book and your programs become more complex, you'll see that debugging can be one of the most time-consuming aspects of programming.

To find the cause of runtime errors, you can start by finding the source statement that was running when the program crashed. You can usually do that by studying the error message that's displayed. In the first console in this figure, for example, you can see that the statement at line 20 in the main method of the InvoiceApp class was running when the program crashed. That's the statement that used the nextDouble method of the Scanner object, and that indicates that the problem is invalid input data. In chapter 5, you'll learn how to fix this bug.

To find the cause of incorrect output, you can start by figuring out why the application produced the output that it did. For instance, you can start by asking why the second application in this figure didn't prompt the user to enter any test scores. Once you figure that out, you're well on your way to fixing the bug.

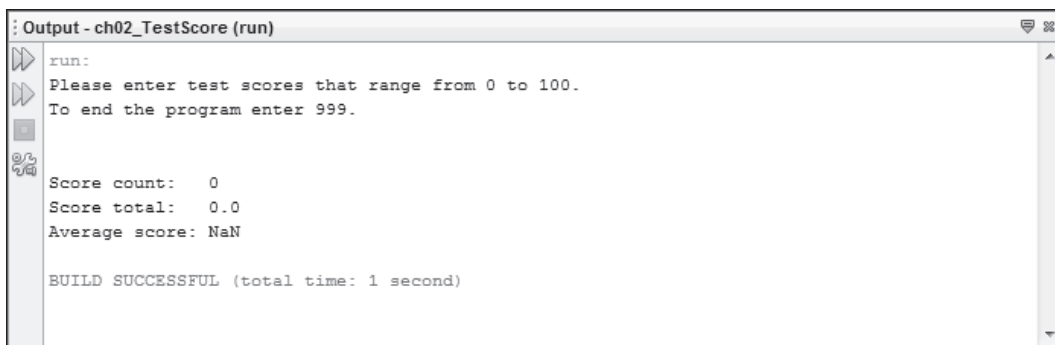## A runtime error that occurred while testing the Invoice application

```
Output - ch02_Invoice (run)

run:
Welcome to the Invoice Total Calculator

Enter subtotal:    $100
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextDouble(Scanner.java:2456)
        at InvoiceApp.main(InvoiceApp.java:20)
Java Result: 1
BUILD SUCCESSFUL (total time: 3 seconds)
```

Incorrect output produced by the Test Score application

```
Output - ch02_TestScore (run)

run:
Please enter test scores that range from 0 to 100.
To end the program enter 999.


Score count:   0
Score total:   0.0
Average score: NaN

BUILD SUCCESSFUL (total time: 1 second)
```

Description

- A *syntax* or *compile-time error* occurs when a statement can't be compiled. Before you can test an application, you must fix the syntax errors.
- To *test* an application, you run it to make sure that it works properly no matter what combinations of valid and invalid data you enter. The goal of testing is to find the errors (or *bugs*) in the application.
- To *debug* an application, you find the causes of the bugs and fix them.
- One type of bug leads to a *runtime error* (also known as a *runtime exception*) that causes the program to end prematurely. This type of bug must be fixed before testing can continue.
- Even if an application runs to completion, the results may be incorrect due to *logic errors*. These bugs must also be fixed.

Debugging tips

- For a runtime error, go to the line in the source code that was running when the program crashed. In NetBeans, you can do that by clicking on the link to the line of source code. That should give you a strong indication of what caused the error.
- For a logical error, first figure out how the source code produced that output. Then, fix the code and test the application again.

Figure 2-19   How to test and debug an application

Perspective

The goal of this chapter has been to get you started with Java programming and to get you started fast. Now, if you understand how the Invoice and Test Score applications in figures 2-17 and 2-18 work, you've come a long way. You should also be able to write comparable programs of your own.

Keep in mind, though, that this chapter is just an introduction to Java programming. So in the next chapter, you'll learn the details about working with data. In chapter 4, you'll learn the details about using control statements. And in chapter 5, you'll learn how to prevent and handle runtime exceptions.

Summary

- The *statements* in a Java program direct the operation of the program. The *comments* document what the program does.

- You must code at least one public *class* for every Java program that you write. The *main method* of this class is executed when you run the class.

- *Variables* are used to store data that changes as a program runs, and you use *assignment statements* to assign values to variables. Two of the most common *data types* for numeric variables are the int and double types.

- A *string* is an object that's created from the String class, and it can contain any characters in the character set. You can use the plus sign to *join* a string with another string or a data type, and you can use assignment statements to *append* one string to another. To include special characters in strings, you can use *escape sequences*.

- When you use a *constructor* to create an *object* from a Java class, you are creating an *instance* of the class. There may be more than one constructor for a class, and a constructor may require one or more *arguments*.

- You *call* a *method* from an object and you call a *static method* from a class. A method may require one or more arguments.

- Before you use many of the classes in the Java *API*, you should code an import statement for the class or for the *package* that contains it.

- You can use the methods of a Scanner object to read data from the *console*, and you can use the print and println methods of the System.out object to print data to the console.

- You can code *if statements* to control the logic of a program based on the true or false values of *Boolean expressions*. You can code *while statements* to repeat a series of statements until a Boolean expression becomes false.

- *Testing* is the process of finding the errors or bugs in an application. *Debugging* is the process of locating and fixing the bugs.

Before you do the exercises for this chapter

If you haven't done it already, you should install and configure the JDK, the NetBeans IDE, and the source code for this book as described in the appendixes.

Exercise 2-1   Test the Invoice application

In this exercise, you'll test the Invoice application that's presented in figure 2-17. That will give you a better idea of how this program works.

1. Start the NetBeans IDE and open the project named ch02_ex1_Invoice. This project should be in this directory:

    ```
    C:\murach\java\netbeans\ex_starts
    ```

2. Open the file named InvoiceApp.java. Review the code for this file and note that NetBeans doesn't display any errors.

3. Test this application with valid subtotal entries like 50, 150, 250, and 1000 so it's easy to see whether or not the calculations are correct.

4. Test the application with a subtotal value like 233.33. This will show that the application doesn't round the results to two decimal places. But in the next chapter, you'll learn how to do that.

5. Test the application with an invalid subtotal value like $1000. This time, the application should crash. Study the error message that's displayed and determine which line of source code in the InvoiceApp class was running when the error occurred. Then, jump to this line by clicking on the link to it.

6. Restart the application, enter a valid subtotal, and enter 20 when the program asks you whether you want to continue. What happens and why?

7. Restart the application and enter two values separated by whitespace (like 1000 20) before pressing the Enter key. What happens and why?

Exercise 2-2   Modify the Test Score application

In this exercise, you'll modify the Test Score application that's presented in figure 2-18. That will give you a chance to write some code of your own.

1. Start NetBeans and open the project named ch02_ex2_TestScore that's in the ex_starts directory shown in the previous exercise.

2. Test this application with valid data to see how it works. Then, test the application with invalid data to see what will cause exceptions. Note that if you enter a test score like 125, the program ends, even though the instructions say that the program ends when you enter 999.

3. Open the file named TestScoreApp.java and modify the while statement so the program only ends when you enter 999. Then, test the program to see how this works.

4. Modify the if statement so it displays an error message like "Invalid entry, not counted" if the user enters a score that's greater than 100 but isn't 999. Then, test this change.

Exercise 2-3   Modify the Invoice application

In this exercise, you'll modify the Invoice application. When you're through with the modifications, a test run should look something like this:

```
Welcome to the Invoice Total Calculator

Enter subtotal:    100
Discount percent: 0.1
Discount amount:  10.0
Invoice total:     90.0

Continue? (y/n): y

Enter subtotal:    500
Discount percent: 0.25
Discount amount:  125.0
Invoice total:     375.0

Continue? (y/n): n

Number of invoices: 2
Average invoice:    232.5
Average discount:   67.5
```

1. Open the project named ch02_ex3_Invoice that's in the ex_starts directory. Then, open the file named InvoiceApp.java.

2. Modify the code so the application ends only when the user enters "n" or "N". As it is now, the application ends when the user enters anything other than "y" or "Y". To do this, you need to use a not operator (!) with the equalsIgnoreCase method. This is illustrated by the third example in figure 2-14. Then, compile this class and test this change by entering 0 at the Continue prompt.

3. Modify the code so it provides a discount of 25 percent when the subtotal is greater than or equal to 500. Then, test this change.

4. Modify the code so it displays the number of invoices, the average invoice amount, and the average discount amount when the user ends the program. Then, test this change.

Exercise 2-4   Use the Java API documentation

This exercise steps you through the Java API documentation for the Scanner, String, and Double classes. That will give you a better idea of how extensive the Java API is.

1. Open a browser and display the Java API documentation as described in figure 2-10.

2. Click the java.util package in the upper left frame and the Scanner class in the lower left frame to display the documentation for the Scanner class. Then, scroll through this documentation to get an idea of its scope.

3. Review the constructors for the Scanner class. The constructor that's presented in this chapter has just an InputStream object as its argument. When you code that argument, remember that System.in represents the InputStream object for the console.

4. Review the methods of the Scanner class with special attention to the next, nextInt, and nextDouble methods. Note that there are three next methods and two nextInt methods. The ones used in this chapter have no arguments. Then, review the has methods in the Scanner class. You'll learn how to use some of these methods in chapter 5.

5. Go to the documentation for the String class, which is in the java.lang package, and note that it offers a number of constructors. In this chapter, though, you learned the shortcut for creating String objects because that's the best way to do that. Now, review the methods for this class with special attention to the equals and equalsIgnoreCase methods.

6. Go to the documentation for the Double class, which is also in the java.lang package. Then, review the static parseDouble and toString methods that you'll learn how to use in the next chapter.

If you find the documentation difficult to follow, rest assured that you'll become comfortable with it before you finish this book. Once you learn how to create your own classes, constructors, and methods, it will make more sense.

How to build your
Java programming skills

The easiest way is to let **Murach's Java Programming** be your guide! So if you've enjoyed this chapter, I hope you'll get your own copy of the book today. And don't miss its companion text for web programming, **Murach's Java Servlets and JSP**. You can use both books to:

Mike Murach, Publisher

- Teach yourself how to code desktop and web applications in Java

- Take advantage of all the time- and work-saving features of the NetBeans IDE as you develop Java applications

- Understand how object-oriented programming really works to create your own 3-tier database applications, the way the pros do

- Pick up a new skill whenever you want or need to by focusing on material that's new to you

- Look up coding details or refresh your memory on forgotten details when you're in the middle of developing a Java application

- Loan to your colleagues who will be asking you more and more questions about Java programming

To get your copy of either or both books, you can order online at **www.murach.com** or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

100% Guarantee

You must be satisfied. Each book you buy directly from us must outperform any competing book or course you've ever tried, or send it back within 90 days for a full refund…no questions asked.

Thanks for your interest in Murach books!

Mike