|  |  |
| --- | --- |
| **Topic** | **Java Control Structures** |
| **Learning Outcomes** | The ability of a computer to perform complex tasks is built on just a few ways of combining simple commands into control structures. In Java, there are just six such structures that are used to determine the normal flow of control in a program—and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the block, the while loop, the do while loop, the for loop, the if statement, and the switch statement. Each of these structures is considered to be a single “statement,” but each is in fact a structured statement that can contain one or more other statements inside itself. At the end of the lesson, you will be able to:1. Know Java Programming Control Structures
2. Understand the Difference between a looping constructs and branching Statements
3. Realize the importance of Control Structures in creating programs
4. Create program using Java Control Structures
 |
| **References** | Lemay, L., Perkins, C. L. (1996), Teach Yourself JAVA in 21 days,  Indianapolis, Indiana, Sams.net.Doug Lowe, Java® All-in-One For Dummies®, 4th Edition), New Jersey,  John Wiley & Sons, Inc.Barry Burd Ph.D., Java® For Dummies®, 7th Edition, New Jersey,  John Wiley & Sons, Inc.David J. Eck, Introduction to Programming Using Java Version 5.0, December 2006 Geneva, NY, Hobart and William Smith CollegesJester Lhee I. Pandio, Computer Programming 1, Global Port Taguig City, STI College |

**Discussion**

**Java Control Structures**

So far, all the programs have run straight through from start to finish without making any decisions along the way. In this chapter, you discover two Java statements that let you create some variety in your programs also you find out how to write programs that do not come and go so quickly. They hang around by using loops, which let them execute the same statements more than once. Also in this chapter we will be using the relational operators discussed in the previous chapter to form the either the loop constructs or branching constructs

**Making Decisions (Java if Statements)**

When you are writing computer programs, you are constantly hitting forks in roads. Did the user correctly type the password? If yes, let the user work; if no, kick the bum out. So, the Java programming language needs a way of making a program branch in one of two directions. Fortunately, the language has a way: It is called an “***if statement.***”

In its most basic form, an “***if statement***” lets you execute a single statement or a block of statements only if a Boolean expression evaluates to true. The basic form of the “***if statement***” looks like this:

if (boolean-expression)statement

Note that the Boolean expression must be enclosed in parentheses. In addition, if you use only a single statement, it must end with a semicolon. However, the statement can also be a statement block enclosed by braces. In that case, each statement within the block needs a semicolon, but the block itself does not

Here’s an example of a typical if statement:

double commissionRate = 0.0;

 if (salesTotal > 10000.0)

commissionRate = 0.05;

In this example, a variable named commissionRate is initialized to 0.0 and then set to 0.05 if salesTotal is greater than 10000.0.

Let us visualized the operation of an if statement using a flowchart.

commissionRate =0.05

double commissionRate = 0.0

Indenting the statement under the “if statement” is customary because it makes the structure of your code more obvious. It is not necessary, but it is always a good idea

No

Sales Total > 10000

Fig. 5.1: Flow Sequence of If Statement

Yes

Here’s an example that uses a block rather than a single statement:

double commissionRate = 0.0;

if (salesTotal > 10000.0){

commissionRate = 0.05;

commission = salesTotal \* commissionRate;

}

If - else statements

An if-else statement adds an additional element to a basic if statement: a statement or block that is executed if the boolean expression is not true. Its basic format is

if (boolean-expression) statement else statement

 Here is an example:

double commissionRate;

if (salesTotal <= 10000.0)

commissionRate = 0.02;

else commissionRate = 0.05;

In this example, the commission rate is set to 2 percent if the sales total is less than or equal to $10,000. If the sales total is greater than $10,000, the commission rate is set to 5 percent. Figure 4-2 shows a flowchart for this if-else statement.

Fig. 5.2: Flow Sequence of If –else Statement

In some cases, you can avoid using the else part of an if-else statement by cleverly rearranging your code. This code has the same effect as the preceding if-else statement:

 double commissionRate = 0.05;

 if (salesTotal <= 10000.0)

 commissionRate = 0.02;

Yes

No

commissionRate =0.02

Sales Total > 10000

commissionRate =0.05

double commissionRate = 0.0

Here is another example of if else if statement

|  |
| --- |
| **Guess the number** |
| import static java.lang.System.out;import java.util.Scanner;import java.util.Random;public class GuessingGame {  public static void main(String args[]) {   Scanner keyboard = new Scanner(System.in);  out.print("Enter an int from 1 to 10: ");  int inputNumber = keyboard.nextInt();  int randomNumber = new Random().nextInt(10) + 1; if (inputNumber == randomNumber) {  out.println("\*\*\*\*\*\*\*\*\*\*");  out.println("\*You win.\*");  out.println("\*\*\*\*\*\*\*\*\*\*"); } else {  out.println("You lose.");  out.print("The random number was ");  out.println(randomNumber + ".");} out.println("Thank you for playing.");  keyboard.close(); } } |
| Program Listing 5.1 |

Nested if statements

The general form of a nested if statement is this:

if (expression-1) // \*\*\*\* ---- Expression 1

 if (expression-2) // \*\*\*\* ---- Expression 2

statement-1

 else

 statement-2

else

 if (expression-3) // \*\*\*\* ---- Expression 3

 statement-3

 else

 statement-4

In this example, expression-1 is first to be evaluated. If it evaluates to true, expression-2 is evaluated. If that expression is true, statement-1 is executed; otherwise statement-2 is executed. But if expression-1 is false, expression-3 is evaluated. If expression-3 is true, statement-3 is executed; otherwise statement-4 is executed.

An if statement that’s contained within another if statement is called an inner if statement, and an if statement that contains another if statement is called an outer if statement. Thus, in the preceding example, the if statement that tests expression-1 is an outer if statement, and the if statements that test expression-2 and expression-3 are inner if statements

Nesting can be as complex as you want, but try to keep it as simple as possible. Also, be sure to use indentation to indicate the structure of the nested statements.

|  |
| --- |
| Table 5.1: Sales and Classes Table |
| Sales | Class 1 | Class 2 |
| 0 to 9,999 | 2% | 2.5% |
| 10,000 and over | 4% | 5% |

Suppose that your company has two classes of sales representatives (Class 1 and Class 2) and that these reps get different commissions for sales below $10,000 and sales above $10,000, according to this table:

You could implement this commission structure with a nested if statement:

 if (salesClass == 1)

 if (salesTotal < 10000.0)

 commissionRate = 0.02;

 else

 commissionRate = 0.04;

 else

 if (salesTotal < 10000.0)

 commissionRate = 0.025;

 else

 commissionRate = 0.05;

This example assumes that if the sales Class variable isn’t 1, it must be 2. If that’s not the case, you have to use an additional if statement for Class 2 sales reps:

Fig. 5.3: Flow Sequence of Nested If Statement

commissionRate =0.025

Yes

No

No

Yes

Yes

SalesClass = 1

double commissionRate = 0.0

commissionRate =0.02

Sales Total <10000

No

commissionRate =0.04

commissionRate =0.05

Sales Total <10000

**else-if statements**

A common pattern for nested if statements is to have a series of if-else statements with another if-else statement in each else part:

if (expression-1)

statement-1

else if (expression-2)

statement-2

else if (expression-3)

statement-3

These statements are sometimes called else-if statements, although that term is unofficial. Officially, all that is going on is that the statement in the else part happens to be another if statement — so this statement is just a type of a nested if statement. It is an especially useful form of nesting, however. Suppose that you want to assign four commission rates based on the sales total, according to this table:

|  |
| --- |
| Table 5.2: Sales and Commission Table |
| Sales | Commission |
| Over $10,000 | 5% |
| $5,000 to $9,999 | 3.5% |
| $1,000 to $4,999 | 2% |
| Under $1,000 | 0% |

You can easily implement a series of else-if statements:

if (salesTotal >= 10000.0)

commissionRate = 0.05;

else if (salesTotal >= 5000.0)

commissionRate = 0.035;

else if (salesTotal >= 1000.0)

commissionRate = 0.02;

else commissionRate = 0.0;

double commissionRate = 0.0

salesTotal >= 10000

Yes

commissionRate =0.05

No

salesTotal >= 5000

Yes

commissionRate =0.035

No

Yes

salesTotal >= 1000

commissionRate =0.02

commissionRate =0.00

No

Fig. 5.4: Flow Sequence for Else - If Statement

You have to think through carefully how you set up these else-if statements.

At first glance, for example, this sequence looks as though it might work:

if (salesTotal > 0.0)

commissionRate = 0.0;

else if (salesTotal >= 1000.0)

commissionRate = 0.02;

else if (salesTotal >= 5000.0)

commissionRate = 0.035;

else if (salesTotal >= 10000.0)

commissionRate = 0.05;

Nice try, but this scenario will not work. These if statements always set the commission rate to 0 percent because the boolean expression in the first if statement always tests true (assuming that the salesTotal isn’t zero or negative — and if it is, none of the other if statements matter). As a result, none of the other if statements are ever evaluated.

**Using the ! operator**

The simplest of the logical operators is Not (!). Technically, it’s a unary prefix operator, which means that you use it with one operand, and you code it immediately in front of that operand.

The Not operator reverses the value of a boolean expression. Thus, if the expression is true, not changes it to false. If the expression is false, not changes it to true.

 Example

double commissionRate = 0.0;

 if (!(salesTotal > 10000.0) )

commissionRate = 0.05;

**The & and && operators**

The & and && operators combine two boolean expressions and return true only if both expressions are true. This type of operation is called an AND operation, because the first expression and the second expression must be true for the AND operator to return true.

Suppose that the sales commission rate should be 2.5% if the sales class is 1 and the sales total is $10,000 or more. You could perform this test with two separate if statements (as I did earlier in this chapter), or you could combine the tests into one if statement:

if ((salesClass == 1) & (salesTotal >= 10000.0))

commissionRate = 0.025;

Here the expressions (salesClass == 1) and (salesTotal >= 10000.0) are evaluated separately. Then the & operator compares the results. If they are both true, the & operator returns true. If one is false or both are false, the & operator returns false.

 Note:

The use of parenthesis in enclosing both conditional statements is important for the & operator be able to determine where the first conditional expression ends and the second conditional expression begins.

The && operator is similar to the & operator, but it leverages your knowledge of logic a bit more. Because both expressions compared by the &operator must be true for the entire expression to be true, there is no reason to evaluate the second expression if the first one returns false. The &operator is not aware of this fact, so it blindly evaluates both expressions before determining the results. The && operator is smart enough to stop when it knows what the outcome is. As a result, usually use && instead of &. Here is the preceding example, and this time it is coded smartly with &&:

if ((salesClass == 1) && (salesTotal >= 10000.0))

commissionRate = 0.025;

**The | and || operators**

The | and || operators are called OR operators because they return true if the first expression is true or if the second expression is true. They also return true if both expressions are true. Suppose that sales representatives get no commission if total sales are less than $1,000 or if the sales class is 3. You could do that with two separate if statements:

if (salesTotal < 1000.0)

commissionRate = 0.0;

if (salesClass == 3)

commissionRate = 0.0;

With an OR operator, however, you can do the same thing with a compound condition:

if ((salesTotal < 1000.0) | (salesClass == 3))

commissionRate = 0.0;

To evaluate the expression for this if statement, Java first evaluates the expressions on either side of the | operator. Then, if at least one of these expressions is true, the whole expression is true. Otherwise the expression is false. In most cases, you should use the Conditional Or operator (||) instead of the regular Or operator (|), like this:

if ((salesTotal < 1000.0) || (salesClass == 3))

commissionRate = 0.0;

Like the Conditional AND operator (&&), the Conditional OR operator stops evaluating as soon as it knows what the outcome is. Suppose that the sales total is $500. Then there is no need to evaluate the second expression. Because the first expression evaluates to true and only one of the expressions needs to be true, Java can skip the second expression. If the sales total is $5,000, of course, the second expression must be evaluated.

**The ^ operator**

The ^ operator performs what in the world of logic is known as an Exclusive OR, commonly abbreviated as XOR. It returns true if one — and only one — of the two subexpressions is true. If both expressions are true, or if both expressions are false, the ^ operator returns false

The ^ operator returns true if the two subexpressions have different results. If they have the same result, it returns false. Suppose that you’re writing software that controls your model railroad set, and you want to find out whether two switches are set in a dangerous position that might allow a collision. If the switches are represented by simple integer variables named switch1 and switch2, and 1 means the track is switched to the left and 2 means the track is switched to the right, you could easily test them like this:

 if ( switch1 == switch2 )

System.out.println ("Trouble! The switches are the same");

else

System.out.println ("OK, the switches are different.");

Now, suppose that (for some reason) one of the switches is represented by an int variable where 1 means the switch goes to the left and any other value means the switch goes to the right, but the other switch is represented by an int variable where –1 means the switch goes to the left and any other value means the switch goes to the right. (Who knows — maybe the switches were made by different manufacturers.)

You could use a compound condition like this:

if (((switch1==1)&&(switch2==-1)) || ((switch1!=1)&&(switch2!=-1)))

System.out.println ("Trouble! The switches are the same");

else

System.out.println ("OK, the switches are different.");

But an XOR operator could do the job with a simpler expression:

if ((switch1==1)^(switch2==-1))

System.out.println ("OK, the switches are different.");

else

System.out.println ("Trouble! The switches are the same");

**The Conditional Operator**

Java has a special operator called the conditional operator that is designed to eliminate the need for if statements in certain situations. It is a ternary operator, which means that it works with three operands. The general form for using the conditional operator is this:

boolean-expression ? expression-1 : expression-2

The boolean expression is evaluated first. If it evaluates to true, expression-1 is evaluated, and the result of this expression becomes the result of the whole expression. If the expression is false, expression-2 is evaluated, and its results are used instead. Suppose that you want to assign a value of 0 to an integer variable named salesTier if total sales are less than $10,000 and a value of 1 if the sales are $10,000 or more. You could do that with this statement:

int tier = salesTotal > 10000.0 ? 1 : 0;

Although not required, a set of parentheses helps make this statement easier to follow:

int tier = (salesTotal > 10000.0) ? 1 : 0;

One common use for the conditional operator is when you are using concatenation to build a text string, and you have a word that may need to be plural based on the value of an integer variable. Suppose that you want to create a string that says "You have x apples", with the value of a variable named appleCount substituted for x. But if apples is 1, the string should be "You have 1 apple", not "You have 1 apples". The following statement does the trick:

String msg = "You have " + appleCount + " apple" + ((appleCount>1) ? "s." : ".");

When Java encounters the ? operator, it evaluates the expression (appleCount>1). If true, it uses the first string (s.). If false, it uses the second string (".").

**Comparing Strings**

Comparing strings in Java takes a little extra care, because the == operator really doesn’t work the way it should. Suppose that you want to know whether a string variable named answer contains the value "Yes". You may be tempted to code an if statement like this:

if (answer == "Yes")

System.out.println ("The answer is Yes.");

Unfortunately, that is not correct. The problem is that in Java, strings are reference types, not primitive types; when you use the == operator with reference types, Java compares the references to the objects, not the objects themselves. As a result, the expression answer == "Yes" doesn’t test whether the value of the string referenced by the answer variable is "Yes". Instead, it tests whether the answer string and the literal string "Yes" point to the same string object in memory. In many cases, they do — but some-times they don’t, and the results are difficult to predict. The correct way to test a string for a given value is to use the equals method of the String class:

if (answer.equals("Yes"))

System.out.println ("The answer is Yes.");

This method actually compares the value of the string object referenced by the variable with the string you pass as a parameter and returns a Boolean result to indicate whether the strings have the same value. The String class has another method, equalsIgnoreCase, that is also useful for comparing strings. It compares strings but ignores case, which is especially useful when you are testing string values entered by users. Suppose that you are writing a program that ends only when the user enters the word End. You could use the equals method to test the string:

if (input.equals("end")) // end the program

In this case, however, the user would have to enter end exactly. If the user enters End or END, the program will not end. It is better to code the if statement like this:

if (input.equalsIgnoreCase("end"))

// end the program

Then the user could end the program by entering the word end spelled with any variation of upper and lowercase letters, including end, End, END, or even eNd.

**Switch Statement**

A common practice in programming in any language is to test a variable against some value, and if it doesn’t match that value, to test it again against a different value, and if it doesn’t match that one to make yet another test, and so on. Using only if statements, this can become unwieldy, depending on how it is formatted and how many different options you have to test.

For example, you might end up with a set of if statements something like this or longer:

if (oper == ‘+’)

addargs(arg1,arg2);

else if (oper == ‘=’)

subargs(arg1,arg2);

else if (oper == ‘\*’)

multargs(arg1,arg2);

else if (oper == ‘/’)

divargs(arg1,arg2);

This form of if statement is called a nested if, because each else statement in turn contains yet another if, and so on, until all possible tests have been made. A common shorthand mechanism for nested ifs that you can use in some cases allows you tests and actions together in a single statement. This is the switch or case statement; in Java it’s switch and behaves as it does in C:

switch (test) {

case value One:

resultOne;

break;

case valueTwo:

resultTwo;

break;

case valueThree:

resultThree; break;

...

default:

defaultresult;

}

In the switch statement, the test (a primitive type of byte, char, short, or int) is compared with each of the case values in turn. If a match is found, the statement, or statements after the test is executed. If no match is found, the default statement is executed. The default is optional, so if there isn’t a match in any of the cases and default doesn’t exist, the switch statement completes without doing anything

Note that the significant limitation of the switch in Java is that the tests and values can be only simple primitive types (and then only primitive types that can be type casted to int). You cannot use larger primitive types (long, float) or objects within a switch, nor can you test for any relationship other than equality. This limits the usefulness of switch to all but the simplest cases; nested ifs can work for any kind of test on any type

Here’s a simple example of a switch statement similar to the nested if shown earlier:

switch (oper) {

case ‘+’:

 addargs(arg1,arg2);

break;

case ‘-’:

subargs(arg1,arg2);

break;

case ‘\*’: multargs(arg1,arg2);

break;

case ‘/’:

divargs(arg1,arg2);

 break;

}

Note the break statement included in every line. Without the explicit break, once a match is made, the statements for that match and also all the statements further down in the switch are executed until a break or the end of the switch is found (and then execution continues after the end of the switch). In some cases, this may be exactly what you want to do, but in most cases, you’ll want to make sure to include the break so that only the statements you want to be executed are executed