CHAPTER 3

Flow of Control

3.1 BRANCHING MECHANISM
if–else Statements
Omitting the else
Compound Statements
Tip: Placing of Braces
Nested Statements
Multiway if–else Statement
Example: State Income Tax
The switch Statement
Pitfall: Forgetting a break in a switch Statement
The Conditional Operator

3.2 BOOLEAN EXPRESSIONS
Simple Boolean Expressions
Pitfall: Using = in Place of ==
Pitfall: Using == with Strings
Lexicographic and Alphabetical Order
Building Boolean Expressions
Evaluating Boolean Expressions
Pitfall: Strings of Inequalities
Tip: Naming boolean Variables
Short-Circuit and Complete Evaluation
Precedence and Associativity Rules

3.3 LOOPS
while Statement and do–while Statement
Algorithms and Pseudocode
Example: Averaging a List of Scores
Tip: End of Input Character
The for Statement
The Comma in for Statements
Tip: Repeat N Times Loops
Pitfall: Extra Semicolon in a for Statement
Pitfall: Infinite Loops
Nested Loops
The break and continue Statements
Loop Bugs
Tracing Variables
Assertion Checks

CHAPTER SUMMARY
ANSWERS TO SELF-TEST EXERCISES
PROGRAMMING PROJECTS
Flow of Control

“If you think we’re wax-works,” he said, “you ought to pay, you know. Wax-works weren’t made to be looked at for nothing. Nohow!”
“Contrariwise,” added the one marked “D EE,” “if you think we’re alive, you ought to speak.”

Lewis Carroll, *Through the Looking-Glass*

**INTRODUCTION**

As in most programming languages, Java handles flow of control with branching and looping statements. Java branching and looping statements are the same as in the C and C++ languages and very similar to what they are in other programming languages. (However, the Boolean expressions that control Java branches and loops are a bit different in Java from what they are in C and C++.)

Most branching and looping statements are controlled by Boolean expressions. A **Boolean expression** is any expression that is either true or false. In Java the primitive type `boolean` has only the two values `true` and `false`, and Boolean expressions evaluate to one of these two values. Before we discuss Boolean expressions and the type `boolean`, we will introduce the Java branching statements using only Boolean expressions whose meaning is intuitively obvious. This will serve to motivate our discussion of Boolean expressions.

**PREREQUISITES**

This chapter uses material from Chapters 1 and 2.

### 3.1 Branching Mechanism

*When you come to a fork in the road, take it.*

Attributed to Yogi Berra

**if-else STATEMENTS**

An if-else statement chooses between two alternative statements based on the value of a Boolean expression. For example, suppose you want to design a program to compute a week’s salary for an hourly employee. Assume the firm
pays an overtime rate of one-and-one-half times the regular rate for all hours after the first 40 hours worked. When the employee works 40 or more hours, the pay is then equal to

\[ \text{rate} \times 40 + 1.5 \times \text{rate} \times (\text{hours} - 40) \]

However, if the employee works less than 40 hours, the correct pay formula is simply

\[ \text{rate} \times \text{hours} \]

The following if-else statement computes the correct pay for an employee whether the employee works less than 40 hours or works 40 or more hours:

```java
if (hours > 40)
    grossPay = rate*40 + 1.5*rate*(hours - 40);
else
    grossPay = rate*hours;
```

The syntax for an if-else statement is given in the box entitled “if-else Statement.” If the Boolean expression in parentheses (after the if) evaluates to true, then the statement before the else is executed. If the Boolean expression evaluates to false, then the statement after the else is executed.

Notice that an if-else statement has smaller statements embedded in it. Most of the statement forms in Java allow you to make larger statements out of smaller statements by combining the smaller statements in certain ways.

Remember that when you use a Boolean expression in an if-else statement, the Boolean expression must be enclosed in parentheses.

### OMITTING THE else

Sometimes you want one of the two alternatives in an if-else statement to do nothing at all. In Java this can be accomplished by omitting the else part. These sorts of statements are referred to as if statements to distinguish them from if-else statements. For example, the first of the following two statements is an if statement:

```java
if (sales > minimum)
    salary = salary + bonus;
System.out.println("salary = \$" + salary);
```

If the value of sales is greater than the value of minimum, the assignment statement is executed and then the following System.out.println statement is executed. On the other hand, if the value of sales is less than or equal to minimum, then the embedded assignment statement is not executed, so the if statement causes no change (that is, no bonus is added to the base salary), and the program proceeds directly to the System.out.println statement.
COMPOUND STATEMENTS

You will often want the branches of an if-else statement to execute more than one statement each. To accomplish this, enclose the statements for each branch between a pair of braces, { and }. A list of statements enclosed in a pair of braces is called a compound statement. A compound statement is treated as a single statement by Java and may be used anywhere that a single statement may be used. Thus, the “Multiple Statement Alternatives” version described in the box entitled “if-else Statement” is really just a special case of the “simple” case with one statement in each branch.

if-else Statement

The if-else statement chooses between two alternative actions based on the value of a Boolean_Expression; that is, an expression that is either true or false, such as balance < 0.

Syntax:

```
if (Boolean_Expression)
  Yes_Statement
else
  No_Statement
```

If the Boolean_Expression is true, then the Yes_Statement is executed. If the Boolean_Expression is false, then the No_Statement is executed.

Example:

```
if (time < limit)
  System.out.println("You made it.");
else
  System.out.println("You missed the deadline.");
```

Omitting the else Part:

You may omit the else part to obtain what is often called an if statement.

Syntax:

```
if (Boolean_Expression)
  Action_Statement
```

If the Boolean_Expression is true, then the Action_Statement is executed; otherwise, nothing happens and the program goes on to the next statement.

Example:

```
if (weight > ideal)
  calorieAllotment = calorieAllotment - 500;
```
There are two commonly used ways of indenting and placing braces in if–else statements. They are illustrated below:

```java
if (myScore > yourScore)
{
    System.out.println("I win!");
    wager = wager + 100;
}
else
{
    System.out.println("I wish these were golf scores.");
    wager = 0;
}
```

The only difference is the placement of braces. We find the first form easier to read and so prefer the first form. The second form saves lines and so some programmers prefer the second form or some minor variant of the second form.

Be sure to note the indenting pattern in these examples.
NESTED STATEMENTS

As you have seen, if-else statements and if statements contain smaller statements within them. Thus far we have used compound statements and simple statements, such as assignment statements, as these smaller substatements, but there are other possibilities. In fact, any statement at all can be used as a subpart of an if-else statement, or other statement that has one or more statements within it.

When nesting statements, you normally indent each level of nested substatements, although there are some special situations (such as a multiway if-else statement) where this rule is not followed.

MULTIWAY if-else STATEMENT

The multiway-if-else statement is not really a different kind of Java statement. It is simply ordinary if-else statements nested inside of if-else statements, but it is thought of as a different kind of statement and is indented differently from other nested statements so as to reflect this thinking.

The syntax for a multiway-if-else statement and a simple example are given in the box entitled “Multiway-if-else Statement.” Note that the Boolean expressions are aligned with one another, and their corresponding actions are also aligned with one another. This makes it easy to see the correspondence between Boolean expressions and actions. The Boolean expressions are evaluated in order until a true Boolean expression is found. At that point the evaluation of Boolean expressions stops, and the action cor-
responding to the first \textit{true} Boolean expression is executed. The final \textit{else} is optional. If there is a final \textit{else} and all the Boolean expressions are \textit{false}, the final action is executed. If there is no final \textit{else} and all the Boolean expressions are \textit{false}, then no action is taken. An example of a multiway-if-else statement is given in the following Programming Example.

\begin{example}
\textbf{STATE INCOME TAX}

Display 3.1 contains a program that uses a multiway if-else statement to compute state income tax. This state computes tax according to the rate schedule below:

1. No tax is paid on the first $15,000 of net income.
2. A tax of 5\% is assessed on each dollar of net income from $15,001 to $30,000.
3. A tax of 10\% is assessed on each dollar of net income over $30,000.

The program uses a multiway-if-else statement with one action for each of the above three cases. The condition for the second case is actually more complicated than it needs to be. The computer will not get to the second condition unless it has already tried the first condition and found it to be \textit{false}. Thus, you know that whenever the computer tries the second condition, it will know that \textit{netIncome} is greater than 15000. Hence, you can replace the line

\begin{verbatim}
else if ((netIncome > 15000) && (netIncome <= 30000))
\end{verbatim}

with the following, and the program will perform exactly the same:

\begin{verbatim}
else if (netIncome <= 30000)
\end{verbatim}

\begin{multiway-if-else}
\textbf{MULTIWAY-if-else STATEMENT}

\textbf{SYNTAX:}

\begin{verbatim}
if (Boolean_Expression_1)
  Statement_1
else if (Boolean_Expression_2)
  Statement_2
  .
  .
else if (Boolean_Expression_n)
  Statement_n
else
  Statement_For_All_Other_Possibilities
\end{verbatim}

\end{multiway-if-else}
EXAMPLE:

```java
if (numberOfPeople < 50)
    System.out.println("Less than 50 people");
else if (numberOfPeople < 100)
    System.out.println("At least 50 and less than 100 people");
else if (numberOfPeople < 200)
    System.out.println("At least 100 and less than 200 people");
else
    System.out.println("At least 200 people");
```

The Boolean expressions are checked in order until the first true Boolean expression is encountered and then the corresponding statement is executed. If none of the Boolean expressions is true, then the `Statement_For_All_Other_Possibilities` is executed.

Display 3.1 Tax Program (Part 1 of 2)

```java
import javax.swing.JOptionPane;

public class IncomeTax
{
    public static void main(String[] args)
    {
        double netIncome, tax, fivePercentTax, tenPercentTax;

        String netIncomeString =
            JOptionPane.showInputDialog("Enter net income.\n" + "Do not include a dollar sign or any commas.");

        netIncome = Double.parseDouble(netIncomeString);

        if (netIncome <= 15000)
            tax = 0;
        else if ((netIncome > 15000) && (netIncome <= 30000))
            //tax = 5% of amount over $15,000
            tax = (0.05*(netIncome - 15000));
        else //netIncome > $30,000
        {
            //fivePercentTax = 5% of income from $15,000 to $30,000.
            fivePercentTax = 0.05*15000;

            //tenPercentTax = 10% of income over $30,000.
            tenPercentTax = 0.10*(netIncome - 30000);

            tax = (fivePercentTax + tenPercentTax);
        }

        JOptionPane.showMessageDialog(null, "Tax due = "$ + tax);
        System.exit(0);
    }
}
```
5. What output will be produced by the following code?

```java
int extra = 2;
if (extra < 0)
    System.out.println("small");
else if (extra > 0)
    System.out.println("large");
else
    System.out.println("medium");
```

6. What would be the output in question 5 if the assignment were changed to the following?

```java
int extra = -37;
```

7. What would be the output in question 5 if the assignment were changed to the following?

```java
int extra = 0;
```

8. Write a multiway if-else statement that classifies the value of an int variable `n` into one of the following categories and writes out an appropriate message:

- `n < 0` or `0 ≤ n < 100` or `n ≥ 100`

Hint: Remember that the Boolean expressions are checked in order.
Chapter 3  Flow of Control

THE switch STATEMENT

The switch statement is the only other kind of Java statement that implements multi-way branches. The syntax for a switch statement and a simple example are shown in the box entitled “The switch Statement.”

When a switch statement is executed, one of a number of different branches is executed. The choice of which branch to execute is determined by a controlling expression given in parentheses after the keyword switch. Following this are a number of occurrences of the reserved word case followed by a constant and a colon. These constants are called case labels. The controlling expression for a switch statement must be of one of the types char, int, short, or byte. The case labels must all be of the same type as the controlling expression. No case label can occur more than once, since that would be an ambiguous instruction. There may also be a section labeled default:, which is usually last.

When the switch statement is executed, the controlling expression is evaluated and the computer looks at the case labels. If it finds a case label that equals the value of the controlling expression, it executes the code for that case label.

The switch statement ends when either a break statement is executed or the end of the switch statement is reached. A break statement consists of the keyword break followed by a semicolon. When the computer executes the statements after a case label, it continues until it reaches a break statement. When the computer encounters a break statement, the switch statement ends. If you omit the break statements, then after executing the code for one case, the computer will go on to execute the code for the next case.

Note that you can have two case labels for the same section of code, as in the following portion of a switch statement:

```java
    case 'A':
    case 'a':
        System.out.println("Excellent. You need not take the final.");
        break;
```

Since the first case has no break statement (in fact, no statement at all), the effect is the same as having two labels for one case, but Java syntax requires one keyword case for each label, such as 'A' and 'a'.

If no case label has a constant that matches the value of the controlling expression, then the statements following the default label are executed. You need not have a default section. If there is no default section and no match is found for the value of the controlling expression, then nothing happens when the switch statement is executed. However, it is safest to always have a default section. If you think your case labels list all possible outcomes, then you can put an error message in the default section.

The default case need not be the last case in a switch statement, but making it the last case, as we have always done, makes the code clearer.
THE switch STATEMENT

SYNTAX:

switch (Controlling_Expression) {
    case Case_Label_1:
        Statement_Sequence_1
        break;
    case Case_Label_2:
        Statement_Sequence_2
        break;
    case Case_Label_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
        break;
}

EXAMPLE:

int vehicleClass;
double toll;

switch (vehicleClass) {
    case 1:
        System.out.println("Passenger car.");
        toll = 0.50;
        break;
    case 2:
        System.out.println("Bus.");
        toll = 1.50;
        break;
    case 3:
        System.out.println("Truck.");
        toll = 2.00;
        break;
    default:
        System.out.println("Unknown vehicle class!");
        break;
}
Chapter 3  Flow of Control

A sample switch statement is shown in Display 3.2. Notice that the case labels need not be listed in order and need not span a complete interval.

**Pitfall**

**FORGETTING A break IN A switch Statement**

If you forget a break in a switch statement, the compiler will not issue an error message. You will have written a syntactically correct switch statement, but it will not do what you intended it to do. Notice the annotation in the example in the box entitled "The switch Statement."

The last case in a switch statement does not need a break, but it is a good idea to include it nonetheless. That way, if a new case is added after the last case, you will not forget to add a break (because it is already there.) This advice about break statements also applies to the default case when it is last. It is best to place the default case last, but that is not required by the Java language, so there is always a possibility of somebody adding a case after the default case.

**Self-Test Exercises**

9. What is the output produced by the following code?

```java
char letter = 'B';
switch (letter)
{
    case 'A':
    case 'a':
        System.out.println("Some kind of A.");
    case 'B':
    case 'b':
        System.out.println("Some kind of B.");
        break;
    default:
    System.out.println("Something else.");
    break;
}
```

10. What output will be produced by the following code?

```java
int key = 1;
switch (key + 1)
{
    case 1:
        System.out.println("Apples");
        break;
    case 2:
        System.out.println("Oranges");
        break;
```
case 3:
    System.out.println("Peaches");
case 4:
    System.out.println("Plums");
    break;
default:
    System.out.println("Fruitless");
}

11. What would be the output in question 10 if the first line were changed to the following?
int key = 3;
12. What would be the output in question 10 if the first line were changed to the following?
int key = 5;

Display 3.2 A switch Statement (Part 1 of 2)
THE CONDITIONAL OPERATOR

You can embed a branch inside of an expression by using a ternary operator known as the conditional operator (also called the ternary operator or arithmetic if). Its use is reminiscent of an older programming style, and we do not advise using it. It is included here for the sake of completeness (and in case you disagree with our programming style).
The conditional operator is a notational variant on certain forms of the *if-else* statement. The following example illustrates the conditional operator. Consider the *if-else* statement:

```java
if (n1 > n2)
    max = n1;
else
    max = n2;
```

This can be expressed using the conditional operator as follows:

```java
max = (n1 > n2) ? n1 : n2;
```

The expression on the right-hand side of the assignment statement is the *conditional operator expression*:

```java
(n1 > n2) ? n1 : n2
```

The `?` and `:` together forms a ternary operator known as the *conditional operator*. A *conditional operator expression* starts with a Boolean expression followed by a `?` and then followed by two expressions separated with a colon. If the Boolean expression is true, then the value of the first of the two expressions is returned as the value of the entire expression; otherwise, the value of the second of the two expressions is returned as the value of the entire expression.

### 3.2 Boolean Expressions

“Contrariwise,” continued Tweedledum, “if it was so, it might be, and if it were so, it would be, but as it isn’t, it ain’t. That’s logic.”

Lewis Carroll, *Through the Looking-Glass*

Now that we have motivated Boolean expressions by using them in *if-else* statements, we go on to discuss them and the type `boolean` in more detail. A *Boolean expression* is simply an expression that is either true or false. The name `Boolean` is derived from George Boole, a 19th-century English logician and mathematician whose work was related to these kinds of expressions.

#### SIMPLE BOOLEAN EXPRESSIONS

We have already been using simple Boolean expressions in *if-else* statements. The simplest Boolean expressions are comparisons of two expressions, such as

```java
time < limit
```
and

\[ \text{balance} \leq 0 \]

A Boolean expression does not need to be enclosed in parentheses to qualify as a Boolean expression, although it does need to be enclosed in parentheses when it is used in an `if-else` statement.

Display 3.3 shows the various Java comparison operators you can use to compare two expressions.

Display 3.3 **Java Comparison Operators**

<table>
<thead>
<tr>
<th>MATH NOTATION</th>
<th>NAME</th>
<th>JAVA NOTATION</th>
<th>JAVA EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
<td>==</td>
<td>( x + 7 == 2 \times y )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer == 'y'</td>
</tr>
<tr>
<td>≠</td>
<td>Not equal to</td>
<td>!=</td>
<td>( \text{score} != 0 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer != 'y'</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;</td>
<td>( \text{time} &gt; \text{limit} )</td>
</tr>
<tr>
<td>≥</td>
<td>Greater than or equal to</td>
<td>=&gt;</td>
<td>( \text{age} \geq 21 )</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;</td>
<td>( \text{pressure} &lt; \text{max} )</td>
</tr>
<tr>
<td>≤</td>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>( \text{time} \leq \text{limit} )</td>
</tr>
</tbody>
</table>

**Pitfall**

**Using = in Place of ==**

Since the equal sign, =, is used for assignment in Java, something else was needed to indicate equality. In Java, equality is indicated with two equal signs with no space between them, as in

```java
if (yourScore == myScore)
    System.out.println("A tie.");
```

Fortunately, if you do use = in place of ==, Java will probably give you a compiler error message. (The only case that does not give an error message is when the expression in parentheses happens to form a correct assignment to a boolean variable.)
Pitfall: Using == with Strings

Although == correctly tests two values of a primitive type, such as two numbers, to see if they are equal, it has a different meaning when applied to objects, such as objects of the class String.1 Recall that an object is something whose type is a class, such as a string. All strings are in the class String (that is, are of type String), so == applied to two strings does not test to see whether the strings are equal. To test two strings (or any two objects) to see if they have equal values, you should use the method equals rather than ==. For example, suppose s1 and s2 are String variables that have been given values, and consider the statement

```
if (s1.equals(s2))
    System.out.println("They are equal strings.");
else
    System.out.println("They are not equal strings.");
```

If s1 and s2 name strings that contain the same characters in the same order, then the output will be "They are equal strings.".

The notation may seem a bit awkward at first, because it is not symmetric between the two things being tested for equality. The two expressions

```
s1.equals(s2)
s2.equals(s1)
```

are equivalent.

The method equalsIgnoreCase behaves similarly to equals, except that with equalsIgnoreCase, the upper- and lowercase versions of the same letter are considered the same. For example, "Hello" and "hello" are not equal because their first characters, 'H' and 'h', are different characters. But they would be considered equal by the method equalsIgnoreCase. For example, the following will output Equal ignoring case.:  

```
if ("Hello".equalsIgnoreCase("hello"))
    System.out.println("Equal ignoring case.");
```

Notice that it is perfectly legal to use a quoted string with a String method, as in the preceding use of equalsIgnoreCase. A quoted string is an object of type String and has all the methods that any other object of type String has.

For the kinds of applications we are looking at in this chapter, you could also use == to test for equality of objects of type String, and it would deliver the correct answer. However, there are situations in which == does not correctly test strings for equality, so you should get in the habit of using equals rather than == to test strings.

---

1 When applied to two strings (or any two objects), == tests to see if they are stored in the same memory location, but we will not discuss that until Chapter 4. For now, we need only note that == does something other than test for the equality of two strings.
Chapter 3  Flow of Control

**THE METHODS equals AND equalsIgnoreCase**

When testing strings for equality, do not use ==. Instead, use either equals or equalsIgnoreCase.

**SYNTAX:**

```java
String.equals(Other_String)
String.equalsIgnoreCase(Other_String)
```

**EXAMPLE:**

```java
String s1;
.
.
if ( s1.equals("Hello") )
    System.out.println("The string is Hello.");
else
    System.out.println("The string is not Hello.");
```

---

**LEXICOGRAPHIC AND ALPHABETICAL ORDER**

The method `compareTo` will test two strings to determine their lexicographic order. **Lexicographic ordering** is similar to alphabetic ordering and is sometimes, but not always, the same as alphabetic ordering. The easiest way to think about lexicographic ordering is to think of it as being the same as alphabetic ordering but with the alphabet ordered differently. Specifically, in lexicographic ordering, the letters and other characters are ordered as in the ASCII ordering, which is shown in Appendix 3.

If `s1` and `s2` are two variables of type `String` that have been given `String` values, then

```java
s1.compareTo(s2)
```

returns a negative number if `s1` comes before `s2` in lexicographic ordering, returns zero if the two strings are equal, and returns a positive number if `s2` comes before `s1`. Thus,

```java
s1.compareTo(s2) < 0
```

returns `true` if `s1` comes before `s2` in lexicographic order and returns `false` otherwise. For example, the following will produce correct output:

```java
if (s1.compareTo(s2) < 0)
    System.out.println(  
        s1 + " precedes " + s2 + " in lexicographic ordering");
else if (s1.compareTo(s2) > 0)
```
Boolean Expressions

System.out.println(s1 + " follows " + s2 + " in lexicographic ordering");
else // s1.compareTo(s2) == 0
    System.out.println(s1 + " equals " + s2);

If you look at the ordering of characters in Appendix 3, you will see that all uppercase letters come before all lowercase letters. For example, 'Z' comes before 'a' in lexicographic order. So when comparing two strings consisting of a mix of lowercase and uppercase letters, lexicographic and alphabetic ordering are not the same. However, as shown in Appendix 3, all the lowercase letters are in alphabetic order. So for any two strings of all lowercase letters, lexicographic order is the same as ordinary alphabetic order. Similarly, in the ordering of Appendix 3, all the uppercase letters are in alphabetic order. So for any two strings of all uppercase letters, lexicographic order is the same as ordinary alphabetic order. Thus, if you treat all uppercase letters as if they were lowercase, then lexicographic ordering becomes the same as alphabetic ordering. This is exactly what the method compareToIgnoreCase does. Thus, the following will produce correct output:

if (s1.compareToIgnoreCase(s2) < 0)
    System.out.println(s1 + " precedes " + s2 + " in ALPHABETIC ordering");
else if (s1.compareToIgnoreCase(s2) > 0)
    System.out.println(s1 + " follows " + s2 + " in ALPHABETIC ordering");
else // s1.compareToIgnoreCase(s2) == 0
    System.out.println(s1 + " equals " + s2 + " IGNORING CASE");

The above code will compile and produce results no matter what characters are in the strings s1 and s2. However, alphabetic order only makes sense, and the output only makes sense, if the two strings consist entirely of letters.

The program in Display 3.4 illustrates some of the string comparisons we have just discussed.

Self-Test Exercises

13. Suppose n1 and n2 are two int variables that have been given values. Write a Boolean expression that returns true if the value of n1 is greater than or equal to the value of n2; otherwise, it returns false.

14. Suppose n1 and n2 are two int variables that have been given values. Write an if-else statement that outputs "n1" if n1 is greater than or equal to n2 and outputs "n2" otherwise.

15. Suppose variable1 and variable2 are two variables that have been given values. How do you test whether they are equal when the variables are of type int? How do you test whether they are equal when the variables are of type String?
Chapter 3  Flow of Control

Display 3.4  Comparing Strings

```
public class StringComparisonDemo {
    public static void main(String[] args) {
        String s1 = "Java isn't just for breakfast.";
        String s2 = "JAVA isn't just for breakfast.";
        if (s1.equals(s2))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");
        if (s2.equals(s1))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");
        if (s1.equalsIgnoreCase(s2))
            System.out.println("But the lines are equal, ignoring case.");
        else
            System.out.println("Lines are not equal, even ignoring case.");
        String s3 = "A cup of java is a joy forever.";
        if (s3.compareToIgnoreCase(s1) < 0)
            System.out.println("" + s3 + "" + s1 + "");
        else
            System.out.println("s3 does not precede s1.");
    }
}
```

**Sample Dialogue**

The two lines are not equal.
The two lines are not equal.
But the lines are equal, ignoring case.
"A cup of java is a joy forever."
precedes
"Java isn't just for breakfast."
in alphabetic ordering
**BUILDING BOOLEAN EXPRESSIONS**

You can combine two Boolean expressions using the “and” operator, which is spelled `&&` in Java. For example, the following Boolean expression is true provided `number` is greater than 2 and `number` is less than 7:

\[(\text{number} > 2) \text{ && } (\text{number} < 7)\]

When two Boolean expressions are connected using an `&&`, the entire expression is true, provided both of the smaller Boolean expressions are true; otherwise, the entire expression is false.

### THE “AND” OPERATOR `&&`

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the “and” operator `&&`.

**SYNTAX (FOR A BOOLEAN EXPRESSION USING `&&`):**

\[(\text{Boolean}_\text{Exp}_1) \text{ && } (\text{Boolean}_\text{Exp}_2)\]

**EXAMPLE (WITHIN AN `if-else` STATEMENT):**

```java
if ( (\text{score} > 0) \text{ && } (\text{score} < 10) )
    \text{System.out.println("score is between 0 and 10.");}
else
    \text{System.out.println("score is not between 0 and 10.");}
```

If the value of `score` is greater than 0 and the value of `score` is also less than 10, then the first `System.out.println` statement will be executed; otherwise, the second `System.out.println` statement will be executed.

16. Assume that `nextWord` is a `String` variable that has been given a `String` value consisting entirely of letters. Write some `Java` code that outputs the message “First half of the alphabet”, provided `nextWord` precedes “N” in alphabetic ordering. If `nextWord` does not precede “N” in alphabetic ordering, it outputs “Second half of the alphabet”. (Note that “N” uses double quotes to produce a `String` value, as opposed to using single quotes to produce a `char` value.)

You can also combine two Boolean expressions using the “or” operator, which is spelled `||` in Java. For example, the following is true provided `count` is less than 3 or `count` is greater than 12:

\[(\text{count} < 3) \text{ || } (\text{count} > 12)\]
Chapter 3  Flow of Control

THE "OR" OPERATOR

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the "or" operator ||.

SYNTAX (FOR A BOOLEAN EXPRESSION USING ||):

(Boolean_Exp_1) || (Boolean_Exp_2)

EXAMPLE (WITHIN AN if-else STATEMENT):

if ((salary > expenses) || (savings > expenses))
    System.out.println("Solvent");
else
    System.out.println("Bankrupt");

When two Boolean expressions are connected using an ||, the entire expression is true, provided that one or both of the smaller Boolean expressions are true; otherwise, the entire expression is false.

You can negate any Boolean expression using the ! operator. If you want to negate a Boolean expression, place the expression in parentheses and place the ! operator in front of it. For example, !(savings < debt) means "savings is not less than debt." The ! operator can usually be avoided. For example,

!(savings < debt)

is equivalent to savings >= debt. In some cases you can safely omit the parentheses, but the parentheses never do any harm. The exact details on omitting parentheses are given in the subsection entitled "Precedence and Associativity Rules."

Self-Test Exercises

17. Write an if-else statement that outputs the word "Passed" provided the value of the variable exam is greater than or equal to 60 and also the value of the variable programsDone is greater than or equal to 10. Otherwise, the if-else statement outputs the word "Failed". The variables exam and programsDone are both of type int.

18. Write an if-else statement that outputs the word "Emergency" provided the value of the variable pressure is greater than 100 or the value of the variable temperature is greater than or equal to 212. Otherwise, the if-else statement outputs the word "OK". The variables pressure and temperature are both of type int.
Evaluating Boolean Expressions

Boolean expressions are used to control branch and loop statements. However, a Boolean expression has an independent identity apart from any branch statement or loop statement you might use it in. A Boolean expression returns either true or false. A variable of type boolean can store the values true and false. Thus, you can set a variable of type boolean equal to a Boolean expression. For example:

```java
boolean madeIt = (time < limit) && (limit < max);
```

A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated. The only difference is that an arithmetic expression uses operations such as +, *, and /, and produces a number as the final result, whereas a Boolean expression uses relational operations such as ==, <, and Boolean operations such as &&, ||, !, and produces one of the two values true and false as the final result.

First let’s review evaluating an arithmetic expression. The same technique will work in the same way to evaluate Boolean expressions. Consider the following arithmetic expression:

```java
(number + 1) * (number + 3)
```

Assume that the variable number has the value 2. To evaluate this arithmetic expression, you evaluate the two sums to obtain the numbers 3 and 5, then you combine these two numbers 3 and 5 using the * operator to obtain 15 as the final value. Notice that in performing this evaluation, you do not multiply the expressions (number + 1) and (number + 3). Instead, you multiply the values of these expressions. You use 3; you do not use (number + 1). You use 5; you do not use (number + 3).

The computer evaluates Boolean expressions the same way. Subexpressions are evaluated to obtain values, each of which is either true or false. In particular, ==, !=, <, <=, and so forth operate on pairs of any primitive type to produce a Boolean value of true or false. These individual values of true or false are then combined according to the rules in the tables shown in Display 3.5. For example, consider the Boolean expression

```java
!( ( count < 3 ) || (count > 7) )
```
which might be the controlling expression for an if-else statement. Suppose the value of count is 8. In this case (count < 3) evaluates to false and (count > 7) evaluates to true, so the preceding Boolean expression is equivalent to

\[ !\left( false \lor true \right) \]

Consulting the tables for \( \lor \) (which is labeled \textbf{OR}), the computer sees that the expression inside the parentheses evaluates to true. Thus, the computer sees that the entire expression is equivalent to

\[ !(true) \]

Consulting the tables again, the computer sees that \( !(true) \) evaluates to false, and so it concludes that \texttt{false} is the value of the original Boolean expression.

A boolean variable, that is, one of type \texttt{boolean}, can be given the value of a Boolean expression by using an assignment statement, in the same way that you use an assign-
Boolean Expressions

A statement to set the value of an `int` variable or any other type of variable. For example, the following sets the value of the `boolean` variable `isPositive` to `false`:

```java
int number = -5;
boolean isPositive;
isPositive = (number > 0);
```

If you prefer, you can combine the last two lines as follows:

```java
boolean isPositive = (number > 0);
```

The parentheses are not needed, but they do make it a bit easier to read.

Once a `boolean` variable has a value, you can use the `boolean` variable just as you would use any other Boolean expression. For example,

```java
boolean isPositive = (number > 0);
if (isPositive)
    System.out.println("The number is positive.");
else
    System.out.println("The number is negative or zero.");
```

is equivalent to

```java
if (number > 0)
    System.out.println("The number is positive.");
else
    System.out.println("The number is negative or zero.");
```

Of course, this is just a toy example. It is unlikely that anybody would use the first of the preceding two examples, but you might use something like it if the value of `number`, and therefore the value of the `boolean` expression, might change, as in the following code, which could (by some stretch of the imagination) be part of a program to evaluate lottery tickets:

```java
boolean isPositive = (number > 0);
while (number > 0)
{
    System.out.println("Wow!");
    number = number - 1000;
}
```

The `boolean` values are `true` and `false`.

`true` and `false` are predefined constants of type `boolean`. (They must be written in lowercase.) In Java, a Boolean expression evaluates to the `boolean` value `true` when it is satisfied and evaluates to the `boolean` value `false` when it is not satisfied.
if (isPositive)
    System.out.println("Your number is positive.");
else
    System.out.println("Sorry, number is not positive.");
System.out.println("Only positive numbers can win.");

true and false ARE NOT NUMBERS

Many programming languages have traditionally used 1 and 0 for true and false. The latest versions of most languages have changed things so that now most languages have a type like boolean with values for true and false. However, even in these newer language versions, values of type boolean will be automatically converted to integers and vice versa when context requires it. In particular, C++ will automatically make such conversions.

In Java the values true and false are not numbers, nor can they be type cast to any numeric type. Similarly, values of type int cannot be type cast to boolean values.

Tip

NAMING boolean VARIABLES

Name a boolean variable with a statement that will be true when the value of the boolean variable is true, such as isPositive, pressureOK, and so forth. That way you can easily understand the meaning of the boolean variable when it is used in an if-else statement, or other control statement. Avoid names that do not unambiguously describe the meaning of the variable's value. Do not use names like numberSign, pressureStatus, and so forth.

- SHORT-CIRCUIT AND COMPLETE EVALUATION

Java takes an occasional shortcut when evaluating a Boolean expression. Notice that in many cases, you need to evaluate only the first of two subexpressions in a Boolean expression. For example, consider the following:

(savings >= 0) && (dependents > 1)

If savings is negative, then (savings >= 0) is false, and, as you can see in the tables in Display 3.5, when one subexpression in an && expression is false, then the whole expression is false, no matter whether the other expression is true or false. Thus, if we know that the first expression is false, there is no need to evaluate the second expression. A similar thing happens with || expressions. If the first of two expressions joined with the || operator is true, then you know the entire expression is true, whether the second expression is true or false. In some situations, the Java language
Boolean Expressions 119

can and does use these facts to save itself the trouble of evaluating the second subexpression in a logical expression connected with an && or an ||. Java first evaluates the leftmost of the two expressions joined by an && or an ||. If that gives it enough information to determine the final value of the expression (independent of the value of the second expression), then Java does not bother to evaluate the second expression. This method of evaluation is called short-circuit evaluation or lazy evaluation.

Now let’s look at an example using && that illustrates the advantage of short-circuit evaluation, and let’s give the Boolean expression some context by placing it in an if statement:

```java
if ( (kids != 0) && ((pieces/kids) >= 2) )
    System.out.println("Each child may have two pieces!");
```

If the value of kids is not zero, this statement involves no subtleties. However, suppose the value of kids is zero and consider how short-circuit evaluation handles this case. The expression (kids != 0) evaluates to false, so there would be no need to evaluate the second expression. Using short-circuit evaluation, Java says that the entire expression is false, without bothering to evaluate the second expression. This prevents a run-time error, since evaluating the second expression would involve dividing by zero.

Java also allows you to ask for complete evaluation. In complete evaluation, when two expressions are joined by an “and” or an “or,” both subexpressions are always evaluated, and then the truth tables are used to obtain the value of the final expression. To obtain complete evaluation in Java, you use & rather than && for “and” and use | in place of || for “or.”

In most situations, short-circuit evaluation and complete evaluation give the same result, but, as you have just seen, there are times when short-circuit evaluation can avoid a run-time error. There are also some situations in which complete evaluation is preferred, but we will not use those techniques in this book, and so we will always use && and || to obtain short-circuit evaluation.

**PRECEDENCE AND ASSOCIATIVITY RULES**

Boolean expressions (and arithmetic expressions) need not be fully parenthesized. If you omit parentheses, Java follows precedence and associativity rules in place of the missing parentheses. One easy way to think of the process is to think of the computer adding parentheses according to these precedence and associativity rules. Some of the Java precedence and associativity rules are given in Display 3.6. (A complete set of precedence and associativity rules is given in Appendix 2.) The computer uses precedence rules to decide on where to insert parentheses, but the precedence rules do not differentiate between two operators at the same precedence level, in which case it uses the associativity rules to “break the tie.”

If one operator occurs higher on the list than another in the precedence table (Display 3.6), the higher-up one is said to have higher precedence. If one operator has higher precedence than another, the operator of higher precedence is grouped with its
operands (its arguments) before the operator of lower precedence. For example, if the computer is faced with the expression

\[ \text{balance} \times \text{rate} + \text{bonus} \]

it notices that \( \times \) has a higher precedence than \( + \) and so it first groups the \( \times \) and its operands, as follows:

\[ (\text{balance} \times \text{rate}) + \text{bonus} \]
Next it groups the + with its operands to obtain the fully parenthesized expression

\[((\text{balance} \times \text{rate}) + \text{bonus})\]

Sometimes two operators have the same precedence, in which case the parentheses are added using the associativity rules. To illustrate this, let’s consider another example:

\[\text{bonus} + \text{balance} \times \text{rate} / \text{correctionFactor} - \text{penalty}\]

The operators \(\times\) and \(/\) have higher precedence than either + or −, so the \(\times\) and \(/\) are grouped first. But the \(\times\) and \(/\) have equal precedence, so the computer consults the associativity rule for \(\times\) and \(/\), which says they associate from left to right, which means the \(\times\), which is the leftmost of \(\times\) and \(/\), is grouped first. So the computer interprets the expression as

\[\text{bonus} + ((\text{balance} \times \text{rate}) / \text{correctionFactor}) - \text{penalty}\]

which in turn is interpreted as

\[\text{bonus} + ((\text{balance} \times \text{rate}) / \text{correctionFactor}) - \text{penalty}\]

because \(/\) has higher precedence than either + or −.

But, this is still not fully parenthesized. The computer still must choose to group + first or − first. According to the table, + and − have equal precedence, so the computer must use the associativity rules, which say + and − are associated left to right. So, it interprets the expression as

\[(\text{bonus} + ((\text{balance} \times \text{rate}) / \text{correctionFactor})) - \text{penalty}\]

which in turn is interpreted as the following fully parenthesized expression:

\[(\text{bonus} + ((\text{balance} \times \text{rate}) / \text{correctionFactor})) - \text{penalty})\]

As you can see from studying the table in Display 3.6, most binary operators associate from left to right. But, the assignment operators associate from right to left. So,

\[\text{number1} = \text{number2} = \text{number3}\]

means

\[\text{number1} = (\text{number2} = \text{number3})\]

which in turn is interpreted as the following fully parenthesized expression:

\[(\text{number1} = (\text{number2} = \text{number3}))\]

However, this fully parenthesized expression may not look like it means anything until we explain a bit more about the assignment operator.

Although we do not advocate using the assignment operator, =, as part of a complex expression, it is an operator that returns a value, just as + and \(\times\) do. When an assignment
operator, \( = \), is used in an expression, it changes the value of the variable on the left-hand side of the assignment operator and also returns a value, namely the new value of the variable on the left-hand side of the expression. So, \((\text{number2} = \text{number3})\) sets \(\text{number2}\) equal to the value of \(\text{number3}\) and returns the value of \(\text{number3}\). Thus,

\[
\text{number1} = \text{number2} = \text{number3}
\]

which is equivalent to

\[
(\text{number1} = (\text{number2} = \text{number3}))
\]

sets both \(\text{number2}\) and \(\text{number1}\) equal to the value of \(\text{number3}\). It is best to not use assignment statements inside of expressions, although simple chains of assignment operators such as the following are clear and acceptable:

\[
\text{number1} = \text{number2} = \text{number3};
\]

Although we discourage using expressions that combine the assignment operator and other operators in complicated ways, let’s try to parenthesize one just for practice. Consider:

\[
\text{number1} = \text{number2} = \text{number3} + 7 \times \text{factor}
\]

The operator of highest precedence is \(\times\), and the operator of next-highest precedence is \(+\), so this expression is equivalent to

\[
\text{number1} = \text{number2} = (\text{number3} + (7 \times \text{factor}))
\]

which leaves only the assignment operators to group. They associate right to left, so the fully parenthesized equivalent version of our expression is

\[
(\text{number1} = (\text{number2} = (\text{number3} + (7 \times \text{factor}))))
\]

(Note that there is no case where two operators have equal precedence but one associates from left to right while the other associates from right to left. That must be true or else there would be cases with conflicting instructions for inserting parentheses.)

The association of operands with operators is called binding. For example, when parentheses determine which two expressions (two operands) are being added by a particular \(+\) sign, that is called binding the two operands to the \(+\) sign. A fully parenthesized expression accomplishes binding for all the operators in an expression.

These examples should make it clear that it can be risky to depend too heavily on the precedence and associativity rules. It is best to include most parentheses and only omit parentheses in situations where the intended meaning is very obvious, such as a simple combination of \(\times\) and \(+\), or a simple chain of \&\&'s or a simple chain of \(||\)'s. The following examples have some omitted parentheses but their meaning should be clear:

\[
\text{rate} \times \text{time} + \text{lead}
\]

\[
(\text{time} < \text{limit}) \&\& (\text{yourScore} > \text{theirScore}) \&\& (\text{yourScore} > 0)
\]

\[
(\text{expenses} < \text{income}) || (\text{expenses} < \text{savings}) || (\text{creditRating} > 0)
\]
Notice that the precedence rules include both arithmetic operators such as + and * as well as Boolean operators such as && and ||. This is because many expressions combine arithmetic and Boolean operations, as in the following simple example:

\[(\text{number} + 1) > 2 \quad || \quad (\text{number} + 5) < -3\]

If you check the precedence rules given in Display 3.6, you will see that this expression is equivalent to

\[(((\text{number} + 1) > 2) \quad || \quad ((\text{number} + 5) < (-3)))\]

because > and < have higher precedence than ||. In fact, you could omit all the parentheses in the above expression and it would have the same meaning (but would be less clear).

It may seem that once an expression is fully parenthesized, the meaning of the expression is then determined. It would seem that to evaluate the expression, you (or the computer) simply evaluate the inner expressions before the outer ones. So, in

\[(((\text{number} + 1) > 2) \quad || \quad ((\text{number} + 5) < (-3)))\]

first the expressions \((\text{number} + 1), (\text{number} + 5),\) and \((-3)\) are evaluated (in any order) and then the > and < are evaluated and then the || is applied. That happens to work in this simple case. In this case, it does not matter which of \((\text{number} + 1), (\text{number} + 5),\) and \((-3)\) is evaluated first, but in certain other expressions it will be necessary to specify which subexpression is evaluated first. The rules for evaluating a fully parenthesized expression are (and indeed must be) more complicated than just evaluating inner expressions before outer expressions.

For an expression with no side effects, the rule of performing inner parenthesized expressions before outer ones is all you need. That rule will get you through most simple expressions, but for expressions with side effects, you need to learn the rest of the story, which is what we do next.

The complications come from the fact that some expressions have side effects. When we say an expression has side effects, we mean that in addition to returning a value, the expression also changes something, such as the value of a variable. Expressions with the assignment operator have side effects; \(\text{pay} = \text{bonus}\), for example, changes the value of \text{pay}. Increment and decrement operators have side effects; \(++n\) changes the value of \(n\).

In expressions that include operators with side effects, you need more rules.

For example, consider

\[(((\text{result} = (++)n)) + (\text{other} = (2*(++)n)))\]

The parentheses seem to say that you or the computer should first do the two increment operators, \(++)n\) and \(++)n\), but the parentheses do not say which of the two \(++)n\) to do first. If \(n\) has the value 2 and we do the leftmost \(++)n\) first, then the variable \text{result} is set to 3 and the variable \text{other} is set to 8 (and the entire expression evaluates to 11). But if we do the rightmost \(++)n\) first, then \text{other} is set to 6 and \text{result} is set to 4 (and the
entire expression evaluates to 10). We need a rule to determine the order of evaluation when we have a "tie" like this. However, rather than simply adding a rule to break such "ties," Java instead takes a completely different approach.

To evaluate an expression, Java uses the following three rules:

Java first does binding; that is, it first fully parenthesizes the expression using precedence and associativity rules, just as we have outlined.
Then it simply evaluates expressions left to right.
If an operator is waiting for its two (or one or three) operands to be evaluated, then that operator is evaluated as soon as its operands have been evaluated.

We'll first do an example with no side effects and then an example of an expression with side effects. First the simple example. Consider the expression

\[6 + 7 \times n - 12\]

and assume the value of \(n\) is 2. Using the precedence and associativity rules, we add parentheses one pair at a time as follows:

\[6 + (7 \times n) - 12\]

then

\[(6 + (7 \times n)) - 12\]

and finally the fully parenthesized version

\[((6 + (7 \times n)) - 12)\]

Next, we evaluate subexpressions left to right. (6 evaluates to 6 and 7 evaluates to 7, but that's so obvious we will not make a big deal of it.) The variable \(n\) evaluates to 2. (Remember we assumed the value of \(n\) was 2.) So, we can rewrite the expression as

\[((6 + (7 \times 2)) - 12)\]

The \(\times\) is the only operator that has both of its operands evaluated, so it evaluates to 14 to produce

\[((6 + 14) - 12)\]

Now \(+\) has both of its operands evaluated, so \((6 + 14)\) evaluates to 20 to yield

\[(20 - 12)\]

which in turn evaluates to 8. So 8 is the value for the entire expression.

This may seem like more work than it should be, but remember, the computer is following an algorithm and proceeds step by step; it does not get inspired to make simplifying assumptions.
Next, let's consider an expression with side effects. In fact, let's consider the one we fully parenthesized earlier. Consider the following fully parenthesized expression and assume the value of \( n \) is 2:

\[
((\text{result} = (++n)) + (\text{other} = (2*++n)))))
\]

Subexpressions are evaluated left to right. So, \( \text{result} \) is evaluated first. When used with the assignment operator \( = \), a variable simply evaluates to itself. So, \( \text{result} \) is evaluated and waiting. Next, \( ++n \) is evaluated and it returns the value 3. So the expression is now known to be equivalent to

\[
((\text{result} = 3) + (\text{other} = (2*(++n))))
\]

Now the assignment operator \( = \) has its two operands evaluated, so \( \text{result} = 3 \) is evaluated. Evaluating \( \text{result} = 3 \) sets the value of \( \text{result} \) equal to 3 and returns the value 3. Thus, the expression is now known to be equivalent to

\[
(3 + (\text{other} = (2*(++n))))
\]

(and the side effect of setting \( \text{result} \) equal to 3 has happened). Proceeding left to right, the next thing to evaluate is the variable \( \text{other} \), which simply evaluates to itself, so you need not rewrite anything.

Proceeding left to right, the next subexpression that can be evaluated is \( n \), which evaluates to 3. (Remember \( n \) has already been incremented once, so \( n \) now has the value 3.) Then \( ++ \) has its only argument evaluated, so it is ready to be evaluated. The evaluation of \( ++n \) has the side effect of setting \( n \) equal to 4 and evaluates to 4. So, the entire expression is equivalent to

\[
(3 + (\text{other} = (2*4)))
\]

The only subexpression that has its operands evaluated is \( 2*4 \), so it is evaluated to 8 to produce

\[
(3 + (\text{other} = 8))
\]

Now the assignment operator \( = \) has both of its operands evaluated, so it evaluates to 8 and has the side effect of setting \( \text{other} \) equal to 8. Thus, we know the value of the expression is

\[
(3 + 8)
\]

which evaluates to 11. So, the entire expression evaluates to 11 (and has the side effects of setting \( \text{result} \) equal to 3, setting \( n \) equal to 4, and setting \( \text{other} \) equal to 8).

These rules also allow for method invocations in expressions. For example, in

\[
(\text{++n} > 0) \&\& (\text{s.length()} > \text{n})
\]
Chapter 3  Flow of Control

the variable \( n \) is incremented before \( n \) is compared to \( s.length() \). When we start defining and using more methods, you will see less-contrived examples of expressions that include method invocations.

All of these rules for evaluating expressions are summarized in the box entitled “Rules for Evaluating Expressions.”

**RULES FOR EVALUATING EXPRESSIONS**

Expressions are evaluated as follows:

1. Binding: Determine the equivalent fully parenthesized expression using the precedence and associativity rules.
2. Proceeding left to right, evaluate whatever subexpressions you can evaluate. (These subexpressions will be operands or method arguments. For example, in simple cases they may be numeric constants or variables.)
3. Evaluate each outer operation (and method invocation) as soon as all of its operands (all its arguments) have been evaluated.

**Self-Test Exercises**

19. Determine the value, true or false, of each of the following Boolean expressions, assuming that the value of the variable count is 0 and the value of the variable limit is 10. (Give your answer as one of the values true or false.)

   a. \((count == 0) && (limit < 20)\)
   b. \(count == 0 && limit < 20\)
   c. \((limit > 20) || (count < 5)\)
   d. \(! (count == 12)\)
   e. \((count == 1) && (x < y)\)
   f. \((count < 10) || (x < y)\)
   g. \(! ((count < 10) || (x < y)) && (count >= 0)\)
   h. \(((limit/count) > 7) || (limit < 20)\)
   i. \((limit < 20) || ((limit/count) > 7)\)
   j. \(((limit/count) > 7) && (limit < 0)\)
   k. \((limit < 0) && ((limit/count) > 7)\)
Loops

It is not true that life is one damn thing after another—
It's one damn thing over and over.

Edna St. Vincent Millay,
Letter to Arthur Darison Ficke, October 24, 1930

Looping mechanisms in Java are similar to those in other high-level languages. The three Java loop statements are the while statement, the do-while statement, and the for statement. The same terminology is used with Java as with other languages. The code that is repeated in a loop is called the body of the loop. Each repetition of the loop body is called an iteration of the loop.

while STATEMENT AND do-while STATEMENT

The syntax for the while statement and its variant, the do-while statement, is given in the box entitled “Syntax for while and do-while Statements.” In both cases, the multi-statement body is a special case of the loop with a single-statement body. The multi-statement body is a single compound statement. Examples of while and do-while statements are given in Display 3.7.

The important difference between the while and do-while loops involves when the controlling Boolean expression is checked. With a while statement, the Boolean expression is checked before the loop body is executed. If the Boolean expression evaluates to false, then the body is not executed at all. With a do-while statement, the body of the loop is executed first and the Boolean expression is checked after the loop body is executed. Thus, the do-while statement always executes the loop body at least once. After this start-up, the while loop and the do-while loop behave the same. After each iteration of the loop body, the Boolean expression is again checked and if it is true, then the loop is iterated again. If it has changed from true to false, then the loop statement ends.

20. Does the following sequence produce a division by zero?

```java
int j = -1;
if ((j > 0) && (1/(j+1) > 10))
    System.out.println(i);
```

21. Convert the following expression to an equivalent fully parenthesized expression:

```
bonus + day * rate / correctionFactor * newGuy - penalty
```
Chapter 3  Flow of Control

Display 3.7 Demonstration of while Loops and do–while Loops (Part 1 of 2)

```java
public class WhileDemo {
    public static void main(String[] args) {
        int countDown;
        System.out.println("First while loop:");
        countDown = 3;
        while (countDown > 0) {
            System.out.println("Hello");
            countDown = countDown - 1;
        }
        System.out.println("Second while loop:");
        countDown = 0;
        while (countDown > 0) {
            System.out.println("Hello");
            countDown = countDown - 1;
        }
        System.out.println("First do–while loop:");
        countDown = 3;
        do {
            System.out.println("Hello");
            countDown = countDown - 1;
        } while (countDown > 0);
        System.out.println("Second do–while loop:");
        countDown = 0;
        do {
            System.out.println("Hello");
            countDown = countDown - 1;
        } while (countDown > 0);
    }
}
```
The first thing that happens when a while loop is executed is that the controlling Boolean expression is evaluated. If the Boolean expression evaluates to false at that point, then the body of the loop is never executed. It may seem pointless to execute the body of a loop zero times, but that is sometimes the desired action. For example, a while loop is often used to sum a list of numbers, but the list could be empty. To be more specific, a checkbook-balancing program might use a while loop to sum the values of all the checks you have written in a month—but you might take a month's vacation and write no checks at all. In that case, there are zero numbers to sum and so the loop is iterated zero times.

**SYNTAX FOR while AND do-while STATEMENTS**

**A while STATEMENT WITH A SINGLE-STATEMENT BODY:**

```
while (Boolean_Expression)
    Statement
```

**A while STATEMENT WITH A MULTI-STATEMENT BODY:**

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
    ...
    Statement_Last
}
```
Chapter 3  Flow of Control

Self-Test Exercises

22. What is the output produced by the following?

```java
int n = 10;
while (n > 0)
{
    System.out.println(n);
    n = n - 3;
}
```

23. What output would be produced in question 22 if the `>` sign were replaced with `<`?

24. What is the output produced by the following?

```java
int n = 10;
do
{
    System.out.println(n);
    n = n - 3;
} while (n > 0);
```

25. What output would be produced in question 24 if the `>` sign were replaced with `<`?
Loops

26. What is the output produced by the following?

```java
int n = -42;
do {
  System.out.println(n);
  n = n - 3;
} while (n > 0);
```

27. What is the most important difference between a while statement and a do-while statement?

Example

**AVERAGING A LIST OF SCORES**

Display 3.8 shows a program that reads in a list of scores and computes their average. It illustrates a number of techniques that are commonly used with loops.

The scores are all nonnegative. This allows the program to use a negative number as an end marker. Note that the negative number is not one of the numbers being averaged in. This sort of end marker is known as a **sentinel value**. A sentinel value need not be a negative number, but it must be some value that cannot occur as a "real" input value. For example, if the input list were a list of even integers, then you could use an odd integer as a sentinel value.

To get the loop to end properly, we want the Boolean expression

```java
next >= 0
```

generated before adding in the number read. This way we avoid adding in the sentinel value. So, we want the loop body to end with

```java
nextString = keyboard.readLine();
next = Double.parseDouble(nextString.trim());
```

To make things work out, this in turn requires that we also place these two lines before the loop. A loop often needs some preliminary statements to set things up before the loop is executed.

ALGORITHMS AND PSEUDOCODE

Dealing with the syntax rules of a programming language is not the hard part of solving a problem with a computer program. The hard part is coming up with the underlying method of solution. This method of solution is called an algorithm. An **algorithm** is a set of precise instructions that leads to a solution. Some approximately equivalent words to algorithm are recipe, method, directions, procedure, and routine.
Chapter 3  Flow of Control

Display 3.8  Averaging a List of Scores (Part 1 of 2)

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Averager {
    public static void main(String[] args) throws IOException {
        BufferedReader keyboard =
            new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter a list of nonnegative scores.");
        System.out.println("One number per line please.");
        System.out.println("Mark the end with a negative number.");
        System.out.println("I will compute their average.");

        double next, sum = 0;
        int count = 0;

        String nextString = keyboard.readLine();
        next = Double.parseDouble(nextString.trim());
        while (next >= 0) {
            sum = sum + next;
            count++;
            nextString = keyboard.readLine();
            next = Double.parseDouble(nextString.trim());
        }

        if (count == 0)
            System.out.println("No scores entered.");
        else {
            double average = sum / count;
            System.out.println(count + " scores read.");
            System.out.println("The average is " + average);
        }
    }
}
```
Loops

An algorithm is normally written in a mixture of a programming language, in our case Java, and English (or other human language). This mixture of programming language and human language is known as pseudocode. Using pseudocode frees you from worrying about fine details of Java syntax so that you can concentrate on the method of solution. Underlying the program in Display 3.8 is an algorithm that can be expressed as the following pseudocode:

Give the user instructions.

\[
\text{count} = 0; \\
\text{sum} = 0; \\
\text{Read a number and store it in a variable named next.} \\
\text{while}(\text{next} \geq 0) \\
\{ \\
\quad \text{sum} = \text{sum} + \text{next}; \\
\quad \text{count}++; \\
\quad \text{Read a number and store it in next.} \\
\}\n\]

The average is \( \frac{\text{sum}}{\text{count}} \) provided \( \text{count} \) is not zero.

Output the results.

Note that when using pseudocode, we do not necessarily declare variables or worry about fine syntax details of Java. The only rule is that the pseudocode be precise and clear enough for a good programmer to convert the pseudocode to syntactically correct Java code.

As you will see, significant programs are written not as a single algorithm, but as a set of interacting algorithms; however, each of these algorithms is normally designed in pseudocode unless the algorithm is exceedingly simple.
Tip

END OF INPUT CHARACTER ✤

The material presented in this Programming Tip uses a detail that is different on different operating systems, and so we will not use this material anywhere in this book outside of this Programming Tip. It is nonetheless a very useful technique.

Display 3.8 averages a list of nonnegative numbers. The numbers are input one per line and the end of the list is indicated by a negative number. This last negative number is known as a sentinel value. This works fine when inputting a list of nonnegative numbers, but what if the list of numbers included positive numbers, negative numbers, and zero. In that case there is no number left to serve as a sentinel value. There is, however, a way to signal the end of keyboard input that does not involve any numbers or any ordinary text. This end of input signal can be used to mark the end of an input list that might have numbers of any kind on the list.

The way you signal the end of a sequence of input lines is to input a certain control character on the last input line. A control character is typed by holding down the Control key while typing in a letter (or other character). For example, to input a Control-Z, you hold down the control key while pressing the Z key. The control key is usually labeled Ctrl or something similar. Different operating systems used different control characters to indicate the end of input. Windows operating systems normally use Control-Z to indicate the end of input. UNIX and Mac operating systems normally use Control-D to indicate the end of input. Let’s call whatever control character is used on your operating system the end of input character.

When the method readLine of the class BufferedReader reads a line consisting of the end of input character, it returns the special value null, which is a defined constant that is part of the Java language. We will say more about null in later chapters, but for what we need now, all you need to know about null is:

- The value null can be stored in a variable of type String, but
- The value null is not equal to any String value.

If nextString is a variable of type String, you use == (not equals) to test if nextString contains null.

Any time you have a list of input lines read by the method readLine of the class BufferedReader, you can use the end of input character as a sentinel value to mark the end of input. For example, if you change the input loop in Display 3.8 to the following, the program will then average a list of any kinds of numbers: positive, negative, or zero:

```java
System.out.println("Enter a list of scores.");
System.out.println("One number per line please.");
System.out.println("Mark the end with the end of input character.");
System.out.println("I will compute their average.");

double next, sum = 0;
int count = 0;
```
The third and final loop statement in Java is the \texttt{for} statement. The \texttt{for} statement is most commonly used to step through some integer variable in equal increments. The \texttt{for} statement is, however, a completely general looping mechanism that can do anything that a \texttt{while} loop can do.

For example, the following \texttt{for} statement sums the integers 1 through 10:

\begin{verbatim}
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
\end{verbatim}

A \texttt{for} statement begins with the keyword \texttt{for} followed by three expressions in parentheses that tell the computer what to do with the controlling variable(s). The beginning of a \texttt{for} statement looks like the following:

\texttt{for (Initialization; Boolean Expression; Update)}
The first expression tells how the variable, variables, or other things are initialized, the second expression gives a Boolean expression that is used to check for when the loop should end, and the last expression tells how the loop control variable or variables are updated after each iteration of the loop body. The loop body is a single statement (typically a compound statement) that follows the heading we just described.

### The for Statement

**Syntax:**

```java
for (Initializing; Boolean_Expression; Update)
    Body
```

The Body may be any Java statement, either a simple statement or, more likely, a compound statement consisting of a list of statements enclosed in braces, `{}`. Notice that the three things in parentheses are separated by two, not three, semicolons.

You are allowed to use any Java expression for the Initializing and the Update expressions, so, you may use more, or fewer, than one variable in the expressions; moreover, the variables may be of any type.

**Example:**

```java
int next, sum = 0;
for (next = 0; next <= 10; next++)
    { sum = sum + next;
      System.out.println("sum up to " + next + " is " + sum);
    }
```

The three expressions at the start of a for statement are separated by two, and only two, semicolons. Do not succumb to the temptation to place a semicolon after the third expression. (The technical explanation is that these three things are expressions, not statements, and so do not require a semicolon at the end.)

A for statement often uses a single int variable to control loop iteration and loop ending. However, the three expressions at the start of a for statement may be any Java expressions and, so, may involve more (or even fewer) than one variable, and the variables may be of any type.

The semantics of the for statement is given in Display 3.9. The syntax for a for statement is given in Display 3.10. Display 3.10 also explains how the for statement can be viewed as a notational variant of the while loop.

A variable may be declared in the heading of a for statement at the same time that it is initialized. For example:

```java
for (int n = 1; n < 10; n++)
    System.out.println(n);
```
Display 3.9 Semantics of the for Statement

for (Initialization; Boolean_Expression; Update)
    Body

Start

Initialization

true

Body

false

Update

End

Display 3.10 for Statement Syntax and Alternate Semantics (Part 1 of 2)

for STATEMENT SYNTAX:

SYNTAX:

for (Initialization; Boolean_Expression; Update)
    Body

EXAMPLE:

for (number = 100; number >= 0; number--)
    System.out.println(number + " bottles of beer on the shelf.");
Chapter 3  Flow of Control

Display 3.10  for Statement Syntax and Alternate Semantics (Part 2 of 2)

**EQUIVALENT while LOOP:**

**EQUIVALENT SYNTAX:**

```
Initialization;
while (Boolean_Expression)
{
    Body
    Update;
}
```

**EQUIVALENT EXAMPLE:**

```java
number = 100;
while (number >= 0)
{
    System.out.println(number + " bottles of beer on the shelf.");
    number--;
}
```

**SAMPLE DIALOGUE**

```
100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
...
0 bottles of beer on the shelf.
```

There are some subtleties to worry about when you declare a variable in the heading of a for statement. These subtleties are discussed in Chapter 4 in the Programming Tip subsection entitled “Declaring Variables in a for Statement.” It might be wise to avoid such declarations within a for statement until you reach Chapter 4, but we mention it here for reference value.

**THE COMMA IN for STATEMENTS**

A for loop can contain multiple initialization actions. Simply separate the actions with commas, as in the following:

```java
for (term = 1, sum = 0; term <= 10; term++)
    sum = sum + term;
```
This for loop has two initializing actions. The variable `term` is initialized to 1 and the variable `sum` is also initialized to 0. Note that you use a comma, not a semicolon, to separate the initialization actions.

You can also use commas to place multiple update actions in a for loop. This can lead to a situation where the for loop has an empty body but still does something useful. For example, the previous for loop can be rewritten to the following equivalent version:

```java
for (term = 1, sum = 0; term <= 10; sum = sum + term, term++)
    //Empty body;
```

This, in effect, makes the loop body part of the update action. We find that it makes for a more readable style if you use the update action only for variables that control the loop, as in the previous version of this for loop. We do not advocate using for loops with no body, but if you do use a for loop with no body, annotate it with a comment such as we did in the preceding for loop. As indicated in the upcoming subsection, “Extra Semicolon in a for Statement,” a for loop with no body can also often occur as the result of a programmer error.

The comma used in a for statement, as we just illustrated, is quite limited in how it can be used. You can use it with assignment statements and with incremented and decremented variables (such as `term++` or `term--`), but not with just any arbitrary statements. In particular, both declaring variables and using the comma in for statements can be troublesome. For example, the following is illegal:

```java
for (int term = 1, double sum = 0; term <= 10; term++)
    sum = sum + term;
```

Even the following is illegal:

```java
double sum;
for (int term = 1, sum = 0; term <= 10; term++)
    sum = sum + term;
```

Java will interpret

```java
int term = 1, sum = 0;
```

as declaring both `term` and `sum` to be int variables and complain that `sum` is already declared.

If you do not declare `sum` anywhere else (and it is acceptable to make `sum` an int variable instead of a double variable), then the following, although we discourage it, is legal:

```java
for (int term = 1, sum = 0; term <= 10; term++)
    sum = sum + term;
```

The first part in parentheses (up to the semicolon) declares both `term` and `sum` to be int variables and initializes both of them.
It is best to simply avoid these possibly confusing examples. When using the comma in a for statement, it's safest to simply declare all variables outside the for statement. If you declare all variables outside the for loop, the rules are no longer complicated.

A for loop can have only one Boolean expression to test for ending the for loop. However, you can perform multiple tests by connecting the tests using && operators to form one larger Boolean expression.

(C, C++, and some other programming languages have a general-purpose comma operator. Readers who have programmed in one of these languages need to be warned that, in Java, there is no comma operator. In Java the comma is a separator, not an operator, and its use is very restricted compared to the comma operator in C and C++.)

**Tip**

**REPEAT N TIMES LOOPS**

The simplest way to produce a loop that repeats the loop body a predetermined number of times is with a for statement. For example, the following is a loop that repeats its loop body three times:

```java
for (int count = 1; count <= 3; count++)
    System.out.println("Hip, Hip, Hurray");
```

The body of a for statement need not make any reference to a loop control variable, like the variable count.

**Pitfall**

**EXTRA SEMICOLON IN A FOR STATEMENT**

You normally do not place a semicolon after the closing parenthesis at the beginning of a for loop. To see what can happen, consider the following for loop:

```java
for (int count = 1; count <= 10; count++);
    System.out.println("Hello");
```

If you did not notice the extra semicolon, you might expect this for loop to write Hello to the screen 10 times. If you do notice the semicolon, you might expect the compiler to issue an error message. Neither of those things happens. If you embed this for loop in a complete program, the compiler will not complain. If you run the program, only one Hello will be output instead of 10 Hellos. What is happening? To answer that question, we need a little background.

One way to create a statement in Java is to put a semicolon after something. If you put a semicolon after number++, you change the expression

```
number++
```
Loops

into the statement

number++;

If you place a semicolon after nothing, you still create a statement. Thus, the semicolon by itself is a statement, which is called the empty statement or the null statement. The empty statement performs no action, but still it is a statement. Therefore, the following is a complete and legitimate for loop, whose body is the empty statement:

```java
for (int count = 1; count <= 10; count++);
```

This for loop is indeed iterated 10 times, but since the body is the empty statement, nothing happens when the body is iterated. This loop does nothing, and it does nothing 10 times! After completing this for loop, the computer goes on to execute the following, which writes Hello to the screen one time:

```java
System.out.println("Hello");
```

This same sort of problem can arise with a while loop. Be careful to not place a semicolon after the closing parenthesis that encloses the Boolean expression at the start of a while loop. A do-while loop has just the opposite problem. You must remember to always end a do-while loop with a semicolon.

**Pitfall**

### INFINITE LOOPS

A while loop, do-while loop, or for loop does not terminate as long as the controlling Boolean expression evaluates to true. This Boolean expression normally contains a variable that will be changed by the loop body, and usually the value of this variable eventually is changed in a way that makes the Boolean expression false and therefore terminates the loop. However, if you make a mistake and write your program so that the Boolean expression is always true, then the loop will run forever. A loop that runs forever is called an infinite loop.

Unfortunately, examples of infinite loops are not hard to come by. First, let’s describe a loop that does terminate. The following Java code will write out the positive even numbers less than 12.

```java
number = 2;
while (number != 12)
{
    System.out.println(number);
    number = number + 2;
}
```
Chapter 3  Flow of Control

NESTED LOOPS

It is perfectly legal to nest one loop statement inside another loop statement. For example, the following nests one for loop inside another for loop:

```java
int rowNum, columnNum;
for (rowNum = 1; rowNum <= 3; rowNum++)
{
    for (columnNum = 1; columnNum <= 2; columnNum++)
    {
        System.out.print(" row " + rowNum + " column " + columnNum);
    }
    System.out.println();
}
```

This produces the following output:

row 1 column 1 row 1 column 2
row 2 column 1 row 2 column 2
row 3 column 1 row 3 column 2
For each iteration of the outer loop, the inner loop is iterated from beginning to end and then one println statement is executed to end the line.

(It is best to avoid nested loops by placing the inner loop inside a method definition and placing a method invocation inside the outer loop. Method definitions are covered in Chapters 4 and 5.)

Self-Test Exercises

28. What is the output of the following?
   ```java
   for (int count = 1; count < 5; count++)
      System.out.print((2 * count) + " ");
   ```

29. What is the output of the following?
   ```java
   for (int n = 10; n > 0; n = n - 2)
      System.out.println("Hello "+ n);
   ```

30. What is the output of the following?
   ```java
   for (double sample = 2; sample > 0; sample = sample - 0.5)
      System.out.print(sample + " ");
   ```

31. Rewrite the following for statement as a while loop (and possibly some additional statements):
   ```java
   int n;
   for (n = 10; n > 0; n = n - 2)
      System.out.println("Hello "+ n);
   ```

32. What is the output of the following loop? Identify the connection between the value of n and the value of the variable log.
   ```java
   int n = 1024;
   int log = 0;
   for (int i = 1; i < n; i = i * 2)
      log++;
   System.out.println(n + " "+ log);
   ```

33. What is the output of the following loop? Comment on the code. (This is not the same as the previous exercise)
   ```java
   int n = 1024;
   int log = 0;
   for (int i = 1; i < n; i = i * 2)
      log++;
   System.out.println(n + " "+ log);
   ```

34. Predict the output of the following nested loops:
   ```java
   int n, m;
   for (n = 1; n <= 10; n++)
      for (m = 10; m >= 1; m--)
         System.out.println(n + " times "+ m + " = " + n*m);
   ```
Chapter 3  Flow of Control

**THE break AND continue STATEMENTS**

In previous subsections, we described the basic flow of control for the `while`, `do-while`, and `for` loops. This is how the loops should normally be used and is the way they are usually used. However, you can alter the flow of control in two ways, and in rare cases these two ways can be useful and safe techniques. The two ways of altering the flow of control are to insert a `break` or `continue` statement. The `break` statement ends the loop. The `continue` statement ends the current iteration of the loop body. The `break` and `continue` statements can be used with any of the Java loop statements.

We described the `break` statement earlier when we discussed the `switch` statement. The `break` statement consists of the keyword `break` followed by a semicolon. When executed, the `break` statement ends the nearest enclosing `switch` or `loop` statement.

The `continue` statement consists of the keyword `continue` followed by a semicolon. When executed, the `continue` statement ends the current loop body iteration of the nearest enclosing `loop` statement.

One point that you should note when using the `continue` statement in a `for` loop is that the `continue` statement transfers control to the update expression. So, any loop control variable will be updated immediately after the `continue` statement is executed.

Note that a `break` statement completely ends the loop. In contrast, a `continue` statement merely ends one loop iteration and the next iteration (if any) continues the loop.

---

35. For each of the following situations, tell which type of loop ( `while`, `do-while`, or `for`) would work best:

a. Summing a series, such as \[\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{10}.\]

b. Reading in the list of exam scores for one student.

c. Reading in the number of days of sick leave taken by employees in a department.

36. What is the output of the following?

```java
int number = 10;
while (number > 0) {
    System.out.println(number);
    number = number + 3;
}
```

37. What is the output of the following?

```java
int n, limit = 10;
for (n = 1; n < limit; n++) {
    System.out.println("n == " + n);
    System.out.println("limit == " + limit);
    limit = n + 2;
}
```
You never absolutely need a break or continue statement. Any code that uses a break or continue statement can be rewritten to do the same thing but not use a break or continue statement. The continue statement can be particularly tricky and can make your code hard to read. It may be best to avoid the continue statement completely or at least use it on only very rare occasions. The use of the break and continue statements in loops is controversial, with many experts saying they should never be used.

You can nest one loop statement inside another loop statement. When doing so, remember that any break or continue statement applies to the innermost loop statement containing the break or continue statement. If there is a switch statement inside a loop, any break statement applies to the innermost loop or switch statement.

Self-Test Exercises

38. What is the output produced by the following?

```java
int number = 10;
while (number > 0)
{
    number = number - 2;
    if (number == 4)
        break;
    System.out.println(number);
} 
System.out.println("The end.");
```

39. What is the output produced by the following?

```java
int number = 10;
while (number > 0)
{
    number = number - 2;
    if (number == 4)
        continue;
    System.out.println(number);
} 
System.out.println("The end.");
```

**LOOP BUGS**

There is a pattern to the kinds of mistakes you are most likely to make when programming with loops. Moreover, there are some standard techniques you can use to locate and fix bugs in your loops.

The two most common kinds of loop errors are unintended infinite loops and off-by-one errors. We have already discussed infinite loops, but we still need to consider off-by-one errors.
Chapter 3  Flow of Control

If your loop has an off-by-one error, that means the loop repeats the loop body one too many times or one too few times. These sorts of errors can result from carelessness in designing a controlling Boolean expression. For example, if you use less-than when you should use less-than-or-equal, this can easily make your loop iterate the body the wrong number of times.

Use of == to test for equality in the controlling Boolean expression of a loop can often lead to an off-by-one error or an infinite loop. This sort of equality testing can work satisfactorily for integers and characters, but is not reliable for floating-point numbers. This is because the floating-point numbers are approximate quantities and == tests for exact equality. The result of such a test is unpredictable. When comparing floating-point numbers, always use something involving less-than or greater-than, such as <=; do not use == or !=. Using == or != to test floating-point numbers can produce an off-by-one error or an unintended infinite loop or even some other type of error. Even when using integer variables, it is best to avoid using == and != and to instead use something involving less-than or greater-than.

Off-by-one errors can easily go unnoticed. If a loop is iterated one too many times, or one too few times, then the results might still look reasonable, but be off by enough to cause trouble later on. Always make a specific check for off-by-one errors by comparing your loop results to results you know to be true by some other means, such as a pencil-and-paper calculation.

TRACING VARIABLES

One good way to discover errors in a loop or any kind of code is to trace some key variables. Tracing variables means watching the variables change value while the program is running. Most programs do not output each variable’s value every time the variable changes, but it can help you to debug your program if you can see all of these variable changes.

Many IDEs (Integrated Development Environments) have a built-in utility that lets you easily trace variables without making any changes to your program. These debugging systems vary from one IDE to another. If you have such a debugging facility, it is worth learning how to use it.

If you do not want to use such a debugging facility, you can trace variables by inserting some temporary output statements in your program. For example, the following code compiles but does still contain an error:

```java
int n = 10;
int sum = 10;
while (n > 1)
{
    sum = sum + n;
    n--;
}
System.out.println("The sum of the integers 1 to 10 is " + sum);
```
To find out what is wrong, you can trace the variables \( n \) and \( \text{sum} \) by inserting output statements as follows:

```java
int n = 10;
int sum = 10;
while (n > 1)
{
    sum = sum + n;
    n--;
    System.out.println("n = "+n); //trace
    System.out.println("sum = "+sum); //trace
}
System.out.println("The sum of the integers 1 to 10 is "+ sum);
```

After you have discovered the error and fixed the bugs in the code, you can remove the trace statements or comment them out by preceding each trace statement with `//`.

### ASSERTION CHECKS

An **assertion** is a sentence that says (asserts) something about the state of your program. An assertion must be a sentence that is either true or false and should be true if there are no mistakes in your program. You can place assertions in your code by making them comments. For example, all the comments in the following code are assertions:

```java
int n = 0;
int sum = 0;
//n == 0 and sum == 0
while (n < 100)
{
    n++;
    sum = sum + n;
    //sum == 1 + 2 + 3 + ... + n
}
//sum == 1 + 2 + 3 + ... + 100
```

Note that each of these assertions can be either true or false, depending on the values of \( n \) and \( \text{sum} \), and they all should be true if the program is performing correctly.

Java has a special statement to check whether an assertion is true. An **assertion check** statement has the following form:

```java
assert Boolean_Expression;
```

If you compile and run your program in the proper ways, the assertion check behaves as follows: If the \( \text{Boolean} \_\text{Expression} \) evaluates to true, nothing happens, but if the \( \text{Boolean} \_\text{Expression} \) evaluates to false, the program ends and outputs an error message saying that an assertion failed.
Chapter 3  Flow of Control

For example, the previously displayed code can be written as follows, with the first comment replaced by an assertion check:

```java
int n = 0;
int sum = 0;
assert (n == 0) && (sum == 0);
while (n < 100)
{
    n++;
    sum = sum + n;
    //sum == 1 + 2 + 3 + ...+ n
}
//sum == 1 + 2 + 3 + ...+ 100
```

Note that we translated only one of the three comments into an assertion check. Not all assertion comments lend themselves to becoming assertion checks. For example, there is no simple way to convert the other two comments into Boolean expressions. Doing so would not be impossible, but you would need to use code that would itself be more complicated than what you would be checking.

---

**Assertion Checking**

An assertion check is a Java statement consisting of the keyword `assert` followed by a Boolean expression and a semicolon. If assertion checking is turned on and the Boolean expression in the assertion check evaluates to `false` when the assertion check is executed, then your program will end and output a suitable error message. If assertion checking is not turned on, then the assertion check is treated as a comment.

**Syntax:**

```java
assert Boolean_Expression;
```

**Example:**

```java
assert (n == 0) && (sum == 0);
```

You can turn assertion checking on and off. When debugging code, you can turn assertion checking on so that a failed assertion will produce an error message. Once your code is debugged, you can turn assertion checking off and your code will run more efficiently.

A program or other class containing assertions must be compiled in a different way, even if you do not intend to run it with assertion checking turned on. After all classes used in a program are compiled, you can run the program with assertion checking either turned on or turned off.
If you compile your classes using a one-line command, you would compile a class with assertion checking as follows:

```
javac -source 1.4 YourProgram.java
```

You can then run your program with assertion checking turned on or off. The normal way of running a program has assertion checking turned off. To run your program with assertion checking turned on, you use the following command:

```
java -enableassertions YourProgram
```

If you are using an IDE, check the documentation for your IDE to see how to handle assertion checking. If you do not find an entry for “assertion checking,” which is likely, check to see how you set compile and run options. (With TextPad you can set things up for assertion checking as follows: On the Configure menu, choose Preferences, then choose Compile Java from the Tools submenu and select the check box for the "Prompt for parameters" option. On the same Tools submenu, choose the Run Java Application command and set the "Prompt for parameters" option for it as well. After you set these preferences, when you compile a class, a window will appear in which you can enter options for the javac compile command (for example, -source 1.4). Similarly, when you run a program, a window will appear in which you can enter options for the java run command (for example, -enableassertions).)

### Self-Test Exercises

40. What is the bug in the code in the subsection “Tracing Variables”?

41. Add some suitable output statements to the following code so that all variables are traced:

```java
int n, sum = 0;
for (n = 1; n < 10; n++)
    sum = sum + n;
System.out.println("1 + 2 + ...+ 9 + 10 == " + sum);
```

42. What is the bug in the following code? What do you call this kind of loop bug?

```java
int n, sum = 0;
for (n = 1; n < 10; n++)
    sum = sum + n;
System.out.println("1 + 2 + ...+ 9 + 10 == " + sum);
```

43. Write an assertion check that checks to see that the value of the variable `time` is less than or equal to the value of the variable `limit`. Both variables are of type `int`.

---

2 If you are running applets, you also need to select the “Prompt for parameters” option for the Run Java Applet command on the Tools submenu.
Chapter 3  Flow of Control

Chapter Summary

- The Java branching statements are the if-else statement and the switch statement.
- A switch statement is a multiway branching statement. You can also form multiway branching statements by nesting if-else statements to form a multiway if-else statement.
- Boolean expressions are evaluated similar to the way arithmetic expressions are evaluated. The value of a Boolean expression can be saved in a variable of type boolean.
- The Java loop statements are the while, do-while, and for statements.
- A do-while statement always iterates its loop body at least one time. Both a while statement and a for statement might iterate its loop body zero times.
- A for loop can be used to obtain the equivalent of the instruction “repeat the loop body n times.”
- Tracing variables is a good method for debugging loops.
- An assertion check can be added to your Java code so that if the assertion is false, then your program halts with an error message.

Answers to Self-Test Exercises

1. if (score > 100)
   System.out.println("High");
   else
     System.out.println("Low");

2. if (savings > expenses)
   { 
     System.out.println("Solvent");
     savings = savings - expenses;
     expenses = 0;
   } 
   else 
   { 
     System.out.println("Bankrupt");
   }

3. if (number > 0)
   System.out.println("Positive");
else
   System.out.println("Not positive");

4. if (salary < deductions)
   { 
     System.out.println("Crazy");
   }
else
{
    System.out.println("OK");
    net = salary - deductions;
}

5.large
6.small
7.medium
8. if (n < 0)
    System.out.println(n + " is less than zero.");
else if (n < 100)
    System.out.println(n + " is between 0 and 99 (inclusive).";
else
    System.out.println(n + " is 100 or larger.");

9. Some kind of B.
10. Oranges
11. Plums
12. Fruitless
13. n1 >= n2
14. if (n1 >= n2)
    System.out.println("n1");
else
    System.out.println("n2");

15. When the variables are of type int, you test for equality using ==, as follows:
    variable1 == variable2

When the variables are of type String, you test for equality using the method equals, as follows:
    variable1.equals(variable2)

In some cases, you might want to use equalsIgnoreCase instead of equals.

16. if (nextWord.compareToIgnoreCase("N") < 0)
    System.out.println("First half of the alphabet");
else
    System.out.println("Second half of the alphabet");

17. if ((exam >= 60) && (programsDone >= 10))
    System.out.println("Passed");
else
    System.out.println("Failed");
Chapter 3  Flow of Control

18.  

\[
\begin{align*}
\text{if ( (pressure > 100) || (temperature >= 212) )} & \\
\text{\quad System.out.println("Emergency");} & \\
\text{else} & \\
\text{\quad System.out.println("OK");}
\end{align*}
\]

19.  

a.  true.

b.  true. Note that expressions a and b mean exactly the same thing. Because the operators == and < have higher precedence than &&, you do not need to include the parentheses. The parentheses do, however, make it easier to read. Most people find the expression in a easier to read than the expression in b, even though they mean the same thing.

c.  true.

d.  true.

e.  false. Since the value of the first subexpression, (count == 1), is false, you know that the entire expression is false without bothering to evaluate the second subexpression. Thus, it does not matter what the values of x and y are. This is called short-circuit evaluation, which is what Java does.

f.  true. Since the value of the first subexpression, (count < 10), is true, you know that the entire expression is true without bothering to evaluate the second subexpression. Thus, it does not matter what the values of x and y are. This is called short-circuit evaluation, which is what Java does.

g.  false. Notice that the expression in g includes the expression in f as a subexpression. This subexpression is evaluated using short-circuit evaluation as we described for f. The entire expression in g is equivalent to

\[
!( (true || (x < y)) && true )
\]

which in turn is equivalent to !(true && true), and that is equivalent to !(true), which is equivalent to the final value of false.

h.  This expression produces an error when it is evaluated because the first subexpression, ((limit/count) > 7), involves a division by zero.

i.  true. Since the value of the first subexpression, (limit < 20), is true, you know that the entire expression is true without bothering to evaluate the second subexpression. Thus, the second subexpression,

\[
((limit/count) > 7)
\]

is never evaluated, so the fact that it involves a division by zero is never noticed by the computer. This is short-circuit evaluation, which is what Java does.

j.  This expression produces an error when it is evaluated because the first subexpression, ((limit/count) > 7), involves a division by zero.
k. false. Since the value of the first subexpression, \((\text{limit} < 0)\), is false, you know that the entire expression is false without bothering to evaluate the second subexpression. Thus, the second subexpression, 
\[
((\text{limit}/\text{count}) > 7)
\]
is never evaluated, so the fact that it involves a division by zero is never noticed by the computer. This is short-circuit evaluation, which is what Java does.

20. No. Since \((j > 0)\) is false and Java uses short-circuit evaluation for &&, the expression \((1/(j+1) > 10)\) is never evaluated.

21. \((\text{bonus} + (((\text{day} \times \text{rate}) / \text{correctionFactor}) \times \text{newGuy}) - \text{penalty})\)

22. 10

7

4

1

23. There will be no output. Since \(n > 0\) is false, the loop body is executed zero times.

24. 10

7

4

1

25. 10

A do-while loop always executes its body at least one time.

26. -42

A do-while loop always executes its body at least one time.

27. With a do-while statement, the loop body is always executed at least once. With a while statement, there can be conditions under which the loop body is not executed at all.

28. 2 4 6 8

29. Hello 10
Hello 8
Hello 6
Hello 4
Hello 2

30. 2.0 1.5 1.0 0.5

31. \(n = 10;\)
\[
\text{while } (n > 0)
\{
\quad \text{System.out.println("Hello "+ n);}
\quad n = n - 2;
\}
\]
32. The output is: 1024 10. The second number is the log to the base 2 of the first number. (If the first number is not a power of two, then only an approximation to the log base 2 is produced.)

33. The output is: 1024 1. The semicolon after the first line of the for loop is probably a pitfall error.

34. The output is too long to reproduce here. The pattern is as follows:

   1 times 10 = 10
   1 times 9 = 9
   .
   .
   1 times 1 = 1
   2 times 10 = 20
   2 times 9 = 18
   .
   .
   2 times 1 = 2
   3 times 10 = 30
   .
   .

35. a. A for loop
   b. and c. Both require a while loop since the input list might be empty. (A for loop also might possibly work, but a do-while loop definitely would not work.)

36. This is an infinite loop. The first few lines of output are

   10
   13
   16
   19
   21

37. This is an infinite loop. The first few lines of output are

   n == 1
   limit == 10;
   n == 2
   limit == 3
   n == 3
   limit == 4
   n == 4
   limit == 5

38. 8
   6
   The end.
39. 8
   6
   2
   0
   The end.

40. If you look at the trace you will see that after one iteration the value of sum is 20. But the value should be $10 + 9$ or 19. This should lead you to think that the variable $n$ is not decremented at the correct time. Indeed, the bug is that the two statements

   \[
   \text{sum} = \text{sum} + n; \\
   n--; \\
   \]

   should be reversed to

   \[
   n--; \\
   \text{sum} = \text{sum} + n; \\
   \]

41. int $n$, sum = 0;
    for ($n = 1; n < 10; n++$)
    {
      System.out.println("$n == " + n + " sum == " + sum);
      //Above line is a trace.
      sum = sum + n;
    }
    System.out.println("After loop"); //trace
    System.out.println("$n == " + n + " sum == " + sum); //trace
    System.out.println("1 + 2 + ... + 9 + 10 == " + sum);

If you study the output of this trace, you will see that 10 is never added in. This is a bug in the loop.

42. This is the code you traced in the previous exercise. If you study the output of this trace, you will see that 10 is never added in. This is an off-by-one error.

43. assert (time <= limit);

Programming Projects

1. It is difficult to make a budget that spans several years, because prices are not stable. If your company needs 200 pencils per year, you cannot simply use this year’s price as the cost of pencils two years from now. Due to inflation the cost is likely to be higher than it is today. Write a program to gauge the expected cost of an item in a specified number of years. The program asks for the cost of the item, the number of years from now that the item will be purchased, and the rate of inflation. The program then outputs the estimated cost of the item after the specified period. Have the user enter the inflation rate as a percentage, like
Chapter 3  Flow of Control

5.6 (percent). Your program should then convert the percent to a fraction, like 0.056, and should use a loop to estimate the price adjusted for inflation.

2. You have just purchased a stereo system that cost $1,000 on the following credit plan: No down payment, an interest rate of 18% per year (and hence 1.5% per month), and monthly payments of $50. The monthly payment of $50 is used to pay the interest and whatever is left is used to pay part of the remaining debt. Hence, the first month you pay 1.5% of $1,000 in interest. That is $15 in interest. So, the remaining $35 is deducted from your debt, which leaves you with a debt of $965.00. The next month you pay interest of 1.5% of $965.00, which is $14.48. Hence, you can deduct $35.52 (which is $50 – $14.48) from the amount you owe. Write a program that will tell you how many months it will take you to pay off the loan, as well as the total amount of interest paid over the life of the loan. Use a loop to calculate the amount of interest and the size of the debt after each month. (Your final program need not output the monthly amount of interest paid and remaining debt, but you may want to write a preliminary version of the program that does output these values.) Use a variable to count the number of loop iterations and hence the number of months until the debt is zero. You may want to use other variables as well. The last payment may be less than $50 if the debt is small, but do not forget the interest. If you owe $50, then your monthly payment of $50 will not pay off your debt, although it will come close. One month’s interest on $50 is only 75 cents.

3. The Fibonacci numbers $F_n$ are defined as follows: $F_0$ is 1, $F_1$ is 1, and

$$F_{i+2} = F_i + F_{i+1}$$

$i = 0, 1, 2, \ldots$. In other words, each number is the sum of the previous two numbers. The first few Fibonacci numbers are 1, 1, 2, 3, 5, and 8. One place where these numbers occur is as certain population growth rates. If a population has no deaths, then the series shows the size of the population after each time period. It takes an organism two time periods to mature to reproducing age, and then the organism reproduces once every time period. The formula applies most straightforwardly to asexual reproduction at a rate of one offspring per time period. In any event, the green crud population grows at this rate and has a time period of five days. Hence, if a green crud population starts out as 10 pounds of crud, then in five days there is still 10 pounds of crud; in ten days there is 20 pounds of crud; in fifteen days 30 pounds, in twenty days 50 pounds, and so forth. Write a program that takes both the initial size of a green crud population (in pounds) and a number of days as input, and outputs the number of pounds of green crud after that many days. Assume that the population size is the same for four days and then increases every fifth day. Your program should allow the user to repeat this calculation as often as desired.

4. The value $e^x$ can be approximated by the sum:

$$1 + x + x^2/2! + x^3/3! + \ldots + x^n/n!$$
Write a program that takes a value \( x \) as input and outputs this sum for \( n \) taken to be each of the values 1 to 10, 50, and 100. Your program should repeat the calculation for new values of \( x \) until the user says she or he is through. The expression \( n! \) is called the factorial of \( n \) and is defined as

\[
n! = 1 \times 2 \times 3 \times \ldots \times n
\]

Use variables of type `double` to store the factorials (or arrange your calculation to avoid any direct calculation of factorials); otherwise, you are likely to produce integer overflow, that is, integers larger than Java allows.