Module 3

Program Control Statements

CRITICAL SKILLS

- **3.1** Input characters from the keyboard
- **3.2** Know the complete form of the **if** statement
- **3.3** Use the **switch** statement
- **3.4** Know the complete form of the **for** loop
- **3.5** Use the **while** loop
- **3.6** Use the **do-while** loop
- **3.7** Use **break** to exit a loop
- **3.8** Use **break** as a form of goto
- **3.9** Apply **continue**
- **3.10** Nest loops

In this module you will learn about the statements that control a program's flow of execution. There are three categories of program control statements: *selection* statements, which include the **if** and the **switch**; *iteration* statements, which include the **for**, **while**, and **do-while** loops; and *jump* statements, which include **break**, **continue**, and **return**. Except for **return**, which is discussed later in this book, the remaining control statements, including the **if** and **for** statements to which you have already had a brief introduction, are examined in detail here. The module begins by explaining how to perform some simple keyboard input.

CRITICAL SKILL 3.1

Input Characters from the Keyboard

Before examining Java's control statements, we will make a short digression that will allow you to begin writing interactive programs. Up to this point, the sample programs in this book have displayed information *to* the user, but they have not received information *from* the user. Thus, you have been using console output, but not console (keyboard) input. The main reason for this is that Java's input system relies upon a rather complex system of classes, the use of which requires an understanding of various features, such as exception handling and classes, that are not discussed until later in this book. There is no direct parallel to the very convenient **println( )** method, for example, that allows you to read various types of data entered by the user. Frankly, Java's approach to console input is not as easy to use as one might like. Also, most real-world Java programs and applets will be graphical and window based, not console based. For these reasons, not much use of console input is found in this book. However, there is one type of console input that *is* easy to use: reading a character from the keyboard. Since several of the examples in this module will make use of this feature, it is discussed here.

The easiest way to read a character from the keyboard is to call **System.in.read( )**. **System.in** is the complement to **System.out**. It is the input object attached to the keyboard. The **read( )** method waits until the user presses a key and then returns the result. The character is returned as an integer, so it must be cast into a **char** to assign it to a **char** variable. By default, console input is line buffered, so you must press ENTER before any character that you type will be sent to your program. Here is a program that reads a character from the keyboard:

```
// Read a character from the keyboard.
class KbIn {
  public static void main(String args[])
    throws java.io.IOException {
```

```
    char ch;

    System.out.print("Press a key followed by ENTER: ");

    ch = (char) System.in.read(); // get a char

    System.out.println("Your key is: " + ch);
  }
}
```

Read a character from the keyboard.

Here is a sample run:

```
Press a key followed by ENTER: t
Your key is: t
```

In the program, notice that **main( )** begins like this:

```
public static void main(String args[])
  throws java.io.IOException {
```

Because **System.in.read( )** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors. It is part of Java's exception handling mechanism, which is discussed in Module 9. For now, don't worry about its precise meaning.

The fact that **System.in** is line buffered is a source of annoyance at times. When you press ENTER, a carriage return, line feed sequence is entered into the input stream. Furthermore, these characters are left pending in the input buffer until you read them. Thus, for some applications, you may need to remove them (by reading them) before the next input operation. You will see an example of this later in this module.

Progress Check

1. What is **System.in**?
2. How can you read a character typed at the keyboard?

---

1. **System.in** is the input object linked to standard input, which is usually the keyboard.
2. To read a character, call **System.in.read( )**.

CRITICAL SKILL 3.2

The if Statement

Module 1 introduced the **if** statement. It is examined in detail here. The complete form of the **if** statement is

if(*condition*) *statement*;
else *statement*;

where the targets of the **if** and **else** are single statements. The **else** clause is optional. The targets of both the **if** and **else** can be blocks of statements. The general form of the **if**, using blocks of statements, is

if(*condition*)
{
  statement sequence
}
else
{
  statement sequence
}

If the conditional expression is true, the target of the **if** will be executed; otherwise, if it exists, the target of the **else** will be executed. At no time will both of them be executed. The conditional expression controlling the **if** must produce a **boolean** result.

To demonstrate the **if** (and several other control statements), we will create and develop a simple computerized guessing game that would be suitable for young children. In the first version of the game, the program asks the player for a letter between A and Z. If the player presses the correct letter on the keyboard, the program responds by printing the message **** Right ****. The program is shown here:

```
// Guess the letter game.
class Guess {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, answer = 'K';

    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");

    ch = (char) System.in.read(); // read a char from the keyboard

    if(ch == answer) System.out.println("** Right **");
  }
}
```

This program prompts the player and then reads a character from the keyboard. Using an **if** statement, it then checks that character against the answer, which is K in this case. If K was entered, the message is displayed. When you try this program, remember that the K must be entered in uppercase.

Taking the guessing game further, the next version uses the **else** to print a message when the wrong letter is picked.

```
// Guess the letter game, 2nd version.
class Guess2 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, answer = 'K';

    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");

    ch = (char) System.in.read(); // get a char

    if(ch == answer) System.out.println("** Right **");
    else System.out.println("...Sorry, you're wrong.");
  }
}
```

Nested ifs

A *nested **if*** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. The main thing to remember about nested **if**s in Java is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and not already associated with an **else**. Here is an example:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // this else refers to if(k > 100)
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j < 20)**, because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i == 10)**. The inner **else** refers to **if(k > 100)**, because it is the closest **if** within the same block.

You can use a nested **if** to add a further improvement to the guessing game. This addition provides the player with feedback about a wrong guess.

```
// Guess the letter game, 3rd version.
class Guess3 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, answer = 'K';

    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");

    ch = (char) System.in.read(); // get a char

    if(ch == answer) System.out.println("** Right **");
    else {
      System.out.print("...Sorry, you're ");

      // a nested if
      if(ch < answer) System.out.println("too low");
      else System.out.println("too high");
    }
  }
}
```

A sample run is shown here:

```
I'm thinking of a letter between A and Z.
Can you guess it: Z
...Sorry, you're too high
```

The if-else-if Ladder

A common programming construct that is based upon the nested **if** is the **if-else-if** *ladder*. It looks like this:

if(*condition*)
  statement;
else if(*condition*)
  statement;
else if(*condition*)
  statement;

.
.
.
else
  statement;

The conditional expressions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the conditions is true, the final **else** statement will be executed. The final **else** often acts as a default condition; that is, if all other conditional tests fail, the last **else** statement is performed. If there is no final **else** and all other conditions are false, no action will take place.

The following program demonstrates the **if-else-if** ladder:

```
// Demonstrate an if-else-if ladder.
class Ladder {
  public static void main(String args[]) {
    int x;

    for(x=0; x<6; x++) {
      if(x==1)
        System.out.println("x is one");
      else if(x==2)
        System.out.println("x is two");
      else if(x==3)
        System.out.println("x is three");
      else if(x==4)
        System.out.println("x is four");
      else
        System.out.println("x is not between 1 and 4"); ← This is the default statement.
    }
  }
}
```

The program produces the following output:

```
x is not between 1 and 4
x is one
x is two
x is three
x is four
x is not between 1 and 4
```

As you can see, the default **else** is executed only if none of the preceding **if** statements succeeds.

Progress Check

1. The condition controlling the **if** must be of what type?
2. To what **if** does an **else** always associate?
3. What is an **if-else-if** ladder?

CRITICAL SKILL 3.3

The switch Statement

The second of Java's selection statements is the **switch**. The **switch** provides for a multiway branch. Thus, it enables a program to select among several alternatives. Although a series of nested **if** statements can perform multiway tests, for many situations the **switch** is a more efficient approach. It works like this: the value of an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general form of the **switch** statement is

```
switch(expression) {
  case constant1:
    statement sequence
    break;
  case constant2:
    statement sequence
    break;
  case constant3:
    statement sequence
    break;
  .
  .
  .
  default:
    statement sequence
}
```

1. The condition controlling an **if** must be of type **boolean**.
2. An **else** always associates with the nearest **if** in the same block that is not already associated with an **else**.
3. An **if-else-if** ladder is a sequence of nested **if-else** statements.

The **switch** expression can be of type **char**, **byte**, **short**, or **int**. (Floating-point expressions, for example, are not allowed.) Frequently, the expression controlling the **switch** is simply a variable. The **case** constants must be literals of a type compatible with the expression. No two **case** constants in the same **switch** can have identical values.

The **default** statement sequence is executed if no **case** constant matches the expression. The **default** is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that **case** are executed until the **break** is encountered or, in the case of **default** or the last **case**, until the end of the **switch** is reached.

The following program demonstrates the **switch**.

```
// Demonstrate the switch.
class SwitchDemo {
  public static void main(String args[]) {
    int i;

    for(i=0; i<10; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero");
          break;
        case 1:
          System.out.println("i is one");
          break;
        case 2:
          System.out.println("i is two");
          break;
        case 3:
          System.out.println("i is three");
          break;
        case 4:
          System.out.println("i is four");
          break;
        default:
          System.out.println("i is five or more");
      }

  }
}
```

The output produced by this program is shown here:

```
i is zero
i is one
i is two
i is three
i is four
i is five or more
i is five or more
i is five or more
i is five or more
i is five or more
```

As you can see, each time through the loop, the statements associated with the **case** constant that matches **i** are executed. All others are bypassed. When **i** is five or greater, no **case** statements match, so the **default** statement is executed.

Technically, the **break** statement is optional, although most applications of the **switch** will use it. When encountered within the statement sequence of a **case**, the **break** statement causes program flow to exit from the entire **switch** statement and resume at the next statement outside the **switch**. However, if a **break** statement does not end the statement sequence associated with a **case**, then all the statements *at and following* the matching **case** will be executed until a **break** (or the end of the **switch**) is encountered.

For example, study the following program carefully. Before looking at the output, can you figure out what it will display on the screen?

```
// Demonstrate the switch without break statements.
class NoBreak {
  public static void main(String args[]) {
    int i;

    for(i=0; i<=5; i++) {
      switch(i) {
        case 0:
          System.out.println("i is less than one");
        case 1:
          System.out.println("i is less than two");
        case 2:
          System.out.println("i is less than three");
        case 3:
          System.out.println("i is less than four");
        case 4:
          System.out.println("i is less than five");
```

The **case** statements fall through here.

```
      }
      System.out.println();
    }
  }
}
```

This program displays the following output:

```
i is less than one
i is less than two
i is less than three
i is less than four
i is less than five

i is less than two
i is less than three
i is less than four
i is less than five

i is less than three
i is less than four
i is less than five

i is less than four
i is less than five

i is less than five
```

As this program illustrates, execution will continue into the next **case** if no **break** statement is present.

You can have empty **case**s, as shown in this example:

```
switch(i) {
  case 1:
  case 2:
  case 3: System.out.println("i is 1, 2 or 3");
    break;
  case 4: System.out.println("i is 4");
    break;
}
```

In this fragment, if **i** has the value 1, 2, or 3, the first **println( )** statement executes. If it is 4, the second **println( )** statement executes. The "stacking" of **case**s, as shown in this example, is common when several **case**s share common code.

Nested switch Statements

It is possible to have a **switch** as part of the statement sequence of an outer **switch**. This is called a nested **switch**. Even if the **case** constants of the inner and outer **switch** contain common values, no conflicts will arise. For example, the following code fragment is perfectly acceptable.

```
switch(ch1) {
  case 'A': System.out.println("This A is part of outer switch.");
    switch(ch2) {
      case 'A':
        System.out.println("This A is part of inner switch");
        break;
      case 'B': // ...
    } // end of inner switch
    break;
  case 'B': // ...
```

Progress Check

1. The expression controlling the **switch** can be of what type?
2. When the **switch** expression matches a **case** constant, what happens?
3. If a **case** sequence does not end in **break**, what happens?

---

1. The **switch** expression can be of type **char**, **short**, **int**, or **byte**.
2. When a matching **case** constant is found, the statement sequence associated with that **case** is executed.
3. If a **case** sequence does not end with **break**, execution continues into the next **case** sequence, if one exists.

Project 3-1 Start Building a Java Help System

Help.java

This project builds a simple help system that displays the syntax for the Java control statements. The program displays a menu containing the control statements and then waits for you to choose one. After one is chosen, the syntax of the statement is displayed. In this first version of the program, help is available for only the **if** and **switch** statements. The other control statements are added in subsequent projects.

Step by Step

1. Create a file called **Help.java**.
2. The program begins by displaying the following menu:

```
Help on:
  1. if
  2. switch
Choose one:
```

To accomplish this, you will use the statement sequence shown here:

```
System.out.println("Help on:");
System.out.println("  1. if");
System.out.println("  2. switch");
System.out.print("Choose one: ");
```

3. Next, the program obtains the user's selection by calling **System.in.read( )**, as shown here:

```
choice = (char) System.in.read();
```

4. Once the selection has been obtained, the program uses the **switch** statement shown here to display the syntax for the selected statement.

```
switch(choice) {
  case '1':
    System.out.println("The if:\n");
    System.out.println("if(condition) statement;");
    System.out.println("else statement;");
    break;
  case '2':
    System.out.println("The switch:\n");
    System.out.println("switch(expression) {");
    System.out.println("  case constant:");
    System.out.println("    statement sequence");
    System.out.println("    break;");
    System.out.println("  // ...");
    System.out.println("}");
```

(continued)

```
    break;
  default:
    System.out.print("Selection not found.");
}
```

Notice how the **default** clause catches invalid choices. For example, if the user enters 3, no **case** constants will match, causing the **default** sequence to execute.

5. Here is the entire **Help.java** program listing:

```
/*
   Project 3-1

   A simple help system.
*/
class Help {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    System.out.println("Help on:");
    System.out.println("  1. if");
    System.out.println("  2. switch");
    System.out.print("Choose one: ");
    choice = (char) System.in.read();

    System.out.println("\n");

    switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
      case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println("  case constant:");
        System.out.println("    statement sequence");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
```

```
            break;
          default:
            System.out.print("Selection not found.");
        }
      }
    }
```

6. Here is a sample run.

```
Help on:
  1. if
  2. switch
Choose one: 1

The if:

if(condition) statement;
else statement;
```

Ask the Expert

Q: Under what conditions should I use an if-else-if **ladder rather than a** switch **when coding a multiway branch?**

A: In general, use an **if-else-if** ladder when the conditions controlling the selection process do not rely upon a single value. For example, consider the following **if-else-if** sequence:

```
if(x < 10) // ...
else if(y != 0) // ...
else if(!done) // ...
```

This sequence cannot be recoded into a **switch** because all three conditions involve different variables—and differing types. What variable would control the **switch**? Also, you will need to use an **if-else-if** ladder when testing floating-point values or other objects that are not of types valid for use in a **switch** expression.

CRITICAL SKILL 3.4

The for Loop

You have been using a simple form of the **for** loop since Module 1. You might be surprised at just how powerful and flexible the **for** loop is. Let's begin by reviewing the basics, starting with the most traditional forms of the **for**.

The general form of the **for** loop for repeating a single statement is

for(*initialization*; *condition*; *iteration*) *statement*;

For repeating a block, the general form is

for(*initialization*; *condition*; *iteration*)
{
  statement sequence
}

The *initialization* is usually an assignment statement that sets the initial value of the *loop control variable,* which acts as the counter that controls the loop. The *condition* is a Boolean expression that determines whether or not the loop will repeat. The *iteration* expression defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The **for** loop will continue to execute as long as the condition tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the **for**.

The following program uses a **for** loop to print the square roots of the numbers between 1 and 99. It also displays the rounding error present for each square root.

```
// Show square roots of 1 to 99 and the rounding error.
class SqrRoot {
  public static void main(String args[]) {
    double num, sroot, rerr;

    for(num = 1.0; num < 100.0; num++) {
      sroot = Math.sqrt(num);
      System.out.println("Square root of " + num +
                         " is " + sroot);

      // compute rounding error
      rerr = num - (sroot * sroot);
      System.out.println("Rounding error is " + rerr);
      System.out.println();
    }
  }
}
```

Notice that the rounding error is computed by squaring the square root of each number. This result is then subtracted from the original number, thus yielding the rounding error.

The **for** loop can proceed in a positive or negative fashion, and it can change the loop control variable by any amount. For example, the following program prints the numbers 100 to –95, in decrements of 5.

```
// A negatively running for loop.
class DecrFor {
  public static void main(String args[]) {
    int x;

    for(x = 100; x > -100; x -= 5)    // Loop control variable is decremented by 5 each time.
      System.out.println(x);
  }
}
```

An important point about **for** loops is that the conditional expression is always tested at the top of the loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with. Here is an example:

```
for(count=10; count < 5; count++)
  x += count; // this statement will not execute
```

This loop will never execute because its control variable, **count**, is greater than 5 when the loop is first entered. This makes the conditional expression, **count < 5**, false from the outset; thus, not even one iteration of the loop will occur.

Some Variations on the for Loop

The **for** is one of the most versatile statements in the Java language because it allows a wide range of variations. For example, multiple loop control variables can be used. Consider the following program:

```
// Use commas in a for statement.
class Comma {
  public static void main(String args[]) {
    int i, j;

    for(i=0, j=10; i < j; i++, j--)    // Notice the two loop control variables.
      System.out.println("i and j: " + i + " " + j);


  }
}
```

The output from the program is shown here:

```
i and j: 0 10
i and j: 1 9
i and j: 2 8
i and j: 3 7
i and j: 4 6
```

Here, commas separate the two initialization statements and the two iteration expressions. When the loop begins, both **i** and **j** are initialized. Each time the loop repeats, **i** is incremented and **j** is decremented. Multiple loop control variables are often convenient and can simplify certain algorithms. You can have any number of initialization and iteration statements, but in practice, more than two or three make the **for** loop unwieldy.

The condition controlling the loop can be any valid Boolean expression. It does not need to involve the loop control variable. In the next example, the loop continues to execute until the user types the letter S at the keyboard.

```
// Loop until an S is typed.
class ForTest {
  public static void main(String args[])
    throws java.io.IOException {

    int i;

    System.out.println("Press S to stop.");

    for(i = 0; (char) System.in.read() != 'S'; i++)
      System.out.println("Pass #" + i);
  }
}
```

Missing Pieces

Some interesting **for** loop variations are created by leaving pieces of the loop definition empty. In Java, it is possible for any or all of the initialization, condition, or iteration portions of the **for** loop to be blank. For example, consider the following program.

```
// Parts of the for can be empty.
class Empty {
  public static void main(String args[]) {
    int i;

    for(i = 0; i < 10; ) {          // <- The iteration expression is missing.
      System.out.println("Pass #" + i);
```

```
      i++; // increment loop control var
    }
  }
}
```

Here, the iteration expression of the **for** is empty. Instead, the loop control variable **i** is incremented inside the body of the loop. This means that each time the loop repeats, **i** is tested to see whether it equals 10, but no further action takes place. Of course, since **i** is still incremented within the body of the loop, the loop runs normally, displaying the following output:

```
Pass #0
Pass #1
Pass #2
Pass #3
Pass #4
Pass #5
Pass #6
Pass #7
Pass #8
Pass #9
```

In the next example, the initialization portion is also moved out of the **for**.

```
// Move more out of the for loop.
class Empty2 {
  public static void main(String args[]) {
    int i;

    i = 0; // move initialization out of loop
    for(; i < 10; ) {
      System.out.println("Pass #" + i);
      i++; // increment loop control var
    }
  }
}
```

The initialization expression is moved out of the loop.

In this version, **i** is initialized before the loop begins, rather than as part of the **for**. Normally, you will want to initialize the loop control variable inside the **for**. Placing the initialization outside of the loop is generally done only when the initial value is derived through a complex process that does not lend itself to containment inside the **for** statement.

The Infinite Loop

You can create an *infinite loop* (a loop that never terminates) using the **for** by leaving the conditional expression empty. For example, the following fragment shows the way most Java programmers create an infinite loop.

```
for(;;) // intentionally infinite loop
{
  //...
}
```

This loop will run forever. Although there are some programming tasks, such as operating system command processors, that require an infinite loop, most "infinite loops" are really just loops with special termination requirements. Near the end of this module you will see how to halt a loop of this type. (Hint: it's done using the **break** statement.)

Loops with No Body

In Java, the body associated with a **for** loop (or any other loop) can be empty. This is because a *null statement* is syntactically valid. Body-less loops are often useful. For example, the following program uses one to sum the numbers 1 through 5.

```
// The body of a loop can be empty.
class Empty3 {
  public static void main(String args[]) {
    int i;
    int sum = 0;

    // sum the numbers through 5
    for(i = 1; i <= 5; sum += i++) ;   <-- No body in this loop!

    System.out.println("Sum is " + sum);
  }
}
```

The output from the program is shown here:

```
Sum is 15
```

Notice that the summation process is handled entirely within the **for** statement, and no body is needed. Pay special attention to the iteration expression:

```
sum += i++
```

Don't be intimidated by statements like this. They are common in professionally written Java programs and are easy to understand if you break them down into their parts. In words, this statement says "add to **sum** the value of **sum** plus **i**, then increment **i**." Thus, it is the same as this sequence of statements:

```
sum = sum + i;
i++;
```

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, the following program computes both the summation and the factorial of the numbers 1 through 5. It declares its loop control variable **i** inside the **for**.

```
// Declare loop control variable inside the for.
class ForVar {
  public static void main(String args[]) {
    int sum = 0;
    int fact = 1;

    // compute the factorial of the numbers through 5
    for(int i = 1; i <= 5; i++) {   ← The variable i is declared inside the for statement.
      sum += i;  // i is known throughout the loop
      fact *= i;
    }

    // but, i is not known here.

    System.out.println("Sum is " + sum);
    System.out.println("Factorial is " + fact);
  }
}
```

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. Thus, in the preceding example, **i** is not accessible outside the **for** loop. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

Before moving on, you might want to experiment with your own variations on the **for** loop. As you will find, it is a fascinating loop.

The Enhanced for Loop

Recently, a new form of the **for** loop, called the *enhanced **for***, was added to Java. The enhanced **for** provides a streamlined way to cycle through the contents of a collection of objects, such as an array. The enhanced **for** loop is discussed in Chapter 5, after arrays have been introduced.

Progress Check

1. Can portions of a **for** statement be empty?
2. Show how to create an infinite loop using **for**.
3. What is the scope of a variable declared within a **for** statement?

CRITICAL SKILL 3.5

The while Loop

Another of Java's loops is the **while**. The general form of the **while** loop is

while(*condition*) *statement*;

where *statement* may be a single statement or a block of statements, and *condition* defines the condition that controls the loop, and it may be any valid Boolean expression. The loop repeats while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

Here is a simple example in which a **while** is used to print the alphabet:

```
// Demonstrate the while loop.
class WhileDemo {
  public static void main(String args[]) {
    char ch;

    // print the alphabet using a while loop
    ch = 'a';
    while(ch <= 'z') {
```

1. Yes. All three parts of the **for**—initialization, condition, and iteration—can be empty.
2. for(;;)
3. The scope of a variable declared within a **for** is limited to the loop. Outside the loop, it is unknown.

```
      System.out.print(ch);
      ch++;
    }
  }
}
```

Here, **ch** is initialized to the letter a. Each time through the loop, **ch** is output and then incremented. This process continues until **ch** is greater than z.

As with the **for** loop, the **while** checks the conditional expression at the top of the loop, which means that the loop code may not execute at all. This eliminates the need for performing a separate test before the loop. The following program illustrates this characteristic of the **while** loop. It computes the integer powers of 2, from 0 to 9.

```
// Compute integer powers of 2.
class Power {
  public static void main(String args[]) {
    int e;
    int result;

    for(int i=0; i < 10; i++) {
      result = 1;
      e = i;
      while(e > 0) {
        result *= 2;
        e--;
      }

      System.out.println("2 to the " + i +
                         " power is " + result);
    }
  }
}
```

The output from the program is shown here:

```
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

Ask the Expert

Q: Given the flexibility inherent in all of Java's loops, what criteria should I use when selecting a loop? That is, how do I choose the right loop for a specific job?

A: Use a **for** loop when performing a known number of iterations. Use the **do-while** when you need a loop that will always perform at least one iteration. The **while** is best used when the loop will repeat an unknown number of times.

Notice that the **while** loop executes only when **e** is greater than 0. Thus, when **e** is zero, as it is in the first iteration of the **for** loop, the **while** loop is skipped.

CRITICAL SKILL 3.6

The do-while Loop

The last of Java's loops is the **do-while**. Unlike the **for** and the **while** loops, in which the condition is tested at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop will always execute at least once. The general form of the **do-while** loop is

do {
  statements;
} while(*condition*);

Although the braces are not necessary when only one statement is present, they are often used to improve readability of the **do-while** construct, thus preventing confusion with the **while**. The **do-while** loop executes as long as the conditional expression is true.

The following program loops until the user enters the letter q.

```
// Demonstrate the do-while loop.
class DWDemo {
  public static void main(String args[])
    throws java.io.IOException {

    char ch;

    do {
      System.out.print("Press a key followed by ENTER: ");
```

```
      ch = (char) System.in.read(); // get a char
    } while(ch != 'q');
  }
}
```

Using a **do-while** loop, we can further improve the guessing game program from earlier in this module. This time, the program loops until you guess the letter.

```
// Guess the letter game, 4th version.
class Guess4 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, answer = 'K';

    do {
      System.out.println("I'm thinking of a letter between A and Z.");
      System.out.print("Can you guess it: ");

      // read a letter, but skip cr/lf
      do {
        ch = (char) System.in.read(); // get a char
      } while(ch == '\n' | ch == '\r');

      if(ch == answer) System.out.println("** Right **");
      else {
        System.out.print("...Sorry, you're ");
        if(ch < answer) System.out.println("too low");
        else System.out.println("too high");
        System.out.println("Try again!\n");
      }
    } while(answer != ch);
  }
}
```

Here is a sample run:

```
I'm thinking of a letter between A and Z.
Can you guess it: A
...Sorry, you're too low
Try again!
```

```
I'm thinking of a letter between A and Z.
Can you guess it: Z
...Sorry, you're too high
Try again!

I'm thinking of a letter between A and Z.
Can you guess it: K
** Right **
```

Notice one other thing of interest in this program. The **do-while** loop shown here obtains the next character, skipping over any carriage return and line feed characters that might be in the input stream:

```
// read a letter, but skip cr/lf
do {
  ch = (char) System.in.read(); // get a char
} while(ch == '\n' | ch == '\r');
```

Here is why this loop is needed: As explained earlier, **System.in** is line buffered—you have to press ENTER before characters are sent. Pressing ENTER causes a carriage return and a line feed character to be generated. These characters are left pending in the input buffer. This loop discards those characters by continuing to read input until neither is present.

Progress Check

1. What is the main difference between the **while** and the **do-while** loops?
2. The condition controlling the **while** can be of any type. True or False?

---

1. The **while** checks its condition at the top of the loop. The **do-while** checks its condition at the bottom of the loop. Thus, a **do-while** will always execute at least once.

2. False. The condition must be of type **boolean**.

Project 3-2 Improve the Java Help System

`Help2.java`

This project expands on the Java help system that was created in Project 3-1. This version adds the syntax for the **for**, **while**, and **do-while** loops. It also checks the user's menu selection, looping until a valid response is entered.

Step by Step

1. Copy **Help.java** to a new file called **Help2.java**.

2. Change the portion of the program that displays the choices so that it uses the loop shown here:

```
do {
  System.out.println("Help on:");
  System.out.println("  1. if");
  System.out.println("  2. switch");
  System.out.println("  3. for");
  System.out.println("  4. while");
  System.out.println("  5. do-while\n");
  System.out.print("Choose one: ");
  do {
    choice = (char) System.in.read();
  } while(choice == '\n' | choice == '\r');
} while( choice < '1' | choice > '5');
```

Notice that a nested **do-while** loop is used to discard any spurious carriage return or line feed characters that may be present in the input stream. After making this change, the program will loop, displaying the menu until the user enters a response that is between 1 and 5.

3. Expand the **switch** statement to include the **for**, **while**, and **do-while** loops, as shown here:

```
switch(choice) {
  case '1':
    System.out.println("The if:\n");
    System.out.println("if(condition) statement;");
    System.out.println("else statement;");
    break;
```

(continued)

```
  case '2':
    System.out.println("The switch:\n");
    System.out.println("switch(expression) {");
    System.out.println("  case constant:");
    System.out.println("    statement sequence");
    System.out.println("    break;");
    System.out.println("  // ...");
    System.out.println("}");
    break;
  case '3':
    System.out.println("The for:\n");
    System.out.print("for(init; condition; iteration)");
    System.out.println(" statement;");
    break;
  case '4':
    System.out.println("The while:\n");
    System.out.println("while(condition) statement;");
    break;
  case '5':
    System.out.println("The do-while:\n");
    System.out.println("do {");
    System.out.println("  statement;");
    System.out.println("} while (condition);");
    break;
}
```

Notice that no **default** statement is present in this version of the **switch**. Since the menu loop ensures that a valid response will be entered, it is no longer necessary to include a **default** statement to handle an invalid choice.

4. Here is the entire **Help2.java** program listing:

```
/*
   Project 3-2

   An improved Help system that uses a
   do-while to process a menu selection.
*/
class Help2 {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    do {
      System.out.println("Help on:");
```

```
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. for");
      System.out.println("  4. while");
      System.out.println("  5. do-while\n");
      System.out.print("Choose one: ");
      do {
        choice = (char) System.in.read();
      } while(choice == '\n' | choice == '\r');
    } while( choice < '1' | choice > '5');

    System.out.println("\n");

    switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
      case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println("  case constant:");
        System.out.println("    statement sequence");
        System.out.println("    break;");
        System.out.println("  // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
      case '4':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
      case '5':
        System.out.println("The do-while:\n");
        System.out.println("do {");
        System.out.println("  statement;");
        System.out.println("} while (condition);");
        break;
    }
  }
}
```

CRITICAL SKILL 3.7

Use break to Exit a Loop

It is possible to force an immediate exit from a loop, bypassing any remaining code in the body of the loop and the loop's conditional test, by using the **break** statement. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakDemo {
  public static void main(String args[]) {
    int num;

    num = 100;

    // loop while i-squared is less than num
    for(int i=0; i < num; i++) {
      if(i*i >= num) break; // terminate loop if i*i >= 100
      System.out.print(i + " ");
    }
    System.out.println("Loop complete.");
  }
}
```

This program generates the following output:

```
0 1 2 3 4 5 6 7 8 9 Loop complete.
```

As you can see, although the **for** loop is designed to run from 0 to **num** (which in this case is 100), the **break** statement causes it to terminate early, when **i** squared is greater than or equal to **num**.

The **break** statement can be used with any of Java's loops, including intentionally infinite loops. For example, the following program simply reads input until the user types the letter q.

```
// Read input until a q is received.
class Break2 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch;

    for( ; ; ) {
      ch = (char) System.in.read(); // get a char
      if(ch == 'q') break;
```

This "infinite" loop is terminated by the **break**.

```
    }
    System.out.println("You pressed q!");
  }
}
```

When used inside a set of nested loops, the **break** statement will break out of only the innermost loop. For example:

```
// Using break with nested loops.
class Break3 {
  public static void main(String args[]) {

    for(int i=0; i<3; i++) {
      System.out.println("Outer loop count: " + i);
      System.out.print("    Inner loop count: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // terminate loop if t is 10
        System.out.print(t + " ");
        t++;
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

```
Outer loop count: 0
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 1
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 2
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

As you can see, the **break** statement in the inner loop causes the termination of only that loop. The outer loop is unaffected.

Here are two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops.

CRITICAL SKILL 3.8

Use break as a Form of goto

In addition to its uses with the **switch** statement and loops, the **break** statement can be employed by itself to provide a "civilized" form of the goto statement. Java does not have a goto statement, because it provides an unstructured way to alter the flow of program execution. Programs that make extensive use of the goto are usually hard to understand and hard to maintain. There are, however, a few places where the goto is a useful and legitimate device. For example, the goto can be helpful when exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems.

The general form of the labeled **break** statement is shown here:

break *label*;

Here, *label* is the name of a label that identifies a block of code. When this form of **break** executes, control is transferred out of the named block of code. The labeled block of code must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control to a block of code that does not enclose the **break** statement.

To name a block, put a label at the start of it. The block being labeled can be a stand-alone block, or a statement that has a block as its target. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks.

```
// Using break with a  label.
class Break4 {
  public static void main(String args[]) {
    int i;

    for(i=1; i<4; i++) {
one:  {
two:    {
three:    {
            System.out.println("\ni is " + i);
```

```
            if(i==1) break one;   // Break to a label.
            if(i==2) break two;
            if(i==3) break three;

            // this is never reached
            System.out.println("won't print");
          }
          System.out.println("After block three.");
        }
        System.out.println("After block two.");
      }
      System.out.println("After block one.");
    }
    System.out.println("After for.");

  }
}
```

The output from the program is shown here:

```
i is 1
After block one.

i is 2
After block two.
After block one.

i is 3
After block three.
After block two.
After block one.
After for.
```

Let's look closely at the program to understand precisely why this output is produced. When **i** is 1, the first **if** statement succeeds, causing a **break** to the end of the block of code defined by label **one**. This causes **After block one.** to print. When **i** is 2, the second **if** succeeds, causing control to be transferred to the end of the block labeled by **two**. This causes the messages **After block two.** and **After block one.** to be printed, in that order. When **i** is 3, the third **if** succeeds, and control is transferred to the end of the block labeled by **three**. Now, all three messages are displayed.

Here is another example. This time, **break** is being used to jump outside of a series of nested **for** loops. When the **break** statement in the inner loop is executed, program control jumps to the end of the block defined by the outer **for** loop, which is labeled by **done**. This causes the remainder of all three loops to be bypassed.

```
// Another example of using break with a label.
class Break5 {
  public static void main(String args[]) {

done:
    for(int i=0; i<10; i++) {
      for(int j=0; j<10; j++) {
        for(int k=0; k<10; k++) {
          System.out.println(k + " ");
          if(k == 5) break done; // jump to done
        }
        System.out.println("After k loop"); // won't execute
      }
      System.out.println("After j loop"); // won't execute
    }
    System.out.println("After i loop");
  }
}
```

The output from the program is shown here:

```
0
1
2
3
4
5
After i loop
```

Precisely where you put a label is very important—especially when working with loops. For example, consider the following program:

```
// Where you put a label is important.
class Break6 {
  public static void main(String args[]) {
    int x=0, y=0;

// here, put label before for statement.
stop1: for(x=0; x < 5; x++) {
         for(y = 0; y < 5; y++) {
```

```
            if(y == 2) break stop1;
            System.out.println("x and y: " + x + " " + y);
          }
        }

        System.out.println();

// now, put label immediately before {
       for(x=0; x < 5; x++)
stop2: {
          for(y = 0; y < 5; y++) {
            if(y == 2) break stop2;
            System.out.println("x and y: " + x + " " + y);
          }
        }

  }
}
```

The output from this program is shown here:

```
x and y: 0 0
x and y: 0 1

x and y: 0 0
x and y: 0 1
x and y: 1 0
x and y: 1 1
x and y: 2 0
x and y: 2 1
x and y: 3 0
x and y: 3 1
x and y: 4 0
x and y: 4 1
```

In the program, both sets of nested loops are the same except for one point. In the first set, the label precedes the outer **for** loop. In this case, when the **break** executes, it transfers control to the end of the entire **for** block, skipping the rest of the outer loop's iterations. In the second set, the label precedes the outer **for**'s opening curly brace. Thus, when **break stop2** executes, control is transferred to the end of the outer **for**'s block, causing the next iteration to occur.

Keep in mind that you cannot **break** to any label that is not defined for an enclosing block. For example, the following program is invalid and will not compile.

```
// This program contains an error.
class BreakErr {
  public static void main(String args[]) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }
  }
}
```

Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer control to that block.

Ask the Expert

Q: You say that the goto **is unstructured and that the break with a label offers a better alternative. But really, doesn't breaking to a label, which might be many lines of code and levels of nesting removed from the** break**, also destructure code?**

A: The short answer is yes! However, in those cases in which a jarring change in program flow is required, breaking to a label still retains some structure. A **goto** has none!

CRITICAL SKILL 3.9

Use continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using **continue**. The **continue** statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop. Thus, **continue** is essentially the complement of **break**. For example, the following program uses **continue** to help print the even numbers between 0 and 100.

```
// Use continue.
class ContDemo {
  public static void main(String args[]) {
    int i;

    // print even numbers between 0 and 100
    for(i = 0; i<=100; i++) {
      if((i%2) != 0) continue; // iterate
      System.out.println(i);
    }
  }
}
```

Only even numbers are printed, because an odd one will cause the loop to iterate early, bypassing the call to **println( )**.

In **while** and **do-while** loops, a **continue** statement will cause control to go directly to the conditional expression and then continue the looping process. In the case of the **for**, the iteration expression of the loop is evaluated, then the conditional expression is executed, and then the loop continues.

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue. Here is an example program that uses **continue** with a label:

```
// Use continue with a label.
class ContToLabel {
  public static void main(String args[]) {

outerloop:
    for(int i=1; i < 10; i++) {
      System.out.print("\nOuter loop pass " + i +
                       ", Inner loop: ");
      for(int j = 1; j < 10; j++) {
        if(j == 5) continue outerloop; // continue outer loop
        System.out.print(j);
      }
    }
  }
}
```

The output from the program is shown here:

```
Outer loop pass 1, Inner loop: 1234
Outer loop pass 2, Inner loop: 1234
```

```
Outer loop pass 3, Inner loop: 1234
Outer loop pass 4, Inner loop: 1234
Outer loop pass 5, Inner loop: 1234
Outer loop pass 6, Inner loop: 1234
Outer loop pass 7, Inner loop: 1234
Outer loop pass 8, Inner loop: 1234
Outer loop pass 9, Inner loop: 1234
```

As the output shows, when the **continue** executes, control passes to the outer loop, skipping the remainder of the inner loop.

Good uses of **continue** are rare. One reason is that Java provides a rich set of loop statements that fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

Progress Check

1. Within a loop, what happens when a **break** (with no label) is executed?
2. What happens when a **break** with a label is executed?
3. What does **continue** do?

---

1. Within a loop, a **break** without a label causes immediate termination of the loop. Execution resumes at the first line of code after the loop.

2. When a labeled **break** is executed, execution resumes at the end of the labeled block.

3. The **continue** statement causes a loop to iterate immediately, bypassing any remaining code. If the **continue** includes a label, the labeled loop is continued.

Project 3-3 Finish the Java Help System

Help3.java

This project puts the finishing touches on the Java help system that was created in the previous projects. This version adds the syntax for **break** and **continue**. It also allows the user to request the syntax for more than one statement. It does this by adding an outer loop that runs until the user enters **q** as a menu selection.

Step by Step

1. Copy **Help2.java** to a new file called **Help3.java**.
2. Surround all of the program code with an infinite **for** loop. Break out of this loop, using **break**, when a letter **q** is entered. Since this loop surrounds all of the program code, breaking out of this loop causes the program to terminate.
3. Change the menu loop as shown here:

```
do {
  System.out.println("Help on:");
  System.out.println("  1. if");
  System.out.println("  2. switch");
  System.out.println("  3. for");
  System.out.println("  4. while");
  System.out.println("  5. do-while");
  System.out.println("  6. break");
  System.out.println("  7. continue\n");
  System.out.print("Choose one (q to quit): ");
  do {
    choice = (char) System.in.read();
  } while(choice == '\n' | choice == '\r');
} while( choice < '1' | choice > '7' & choice != 'q');
```

   Notice that this loop now includes the **break** and **continue** statements. It also accepts the letter **q** as a valid choice.

4. Expand the **switch** statement to include the **break** and **continue** statements, as shown here:

```
case '6':
  System.out.println("The break:\n");
  System.out.println("break; or break label;");
  break;
case '7':
  System.out.println("The continue:\n");
  System.out.println("continue; or continue label;");
  break;
```

(continued)

5. Here is the entire **Help3.java** program listing:

```
/*
   Project 3-3

   The finished Java statement Help system
   that processes multiple requests.
*/
class Help3 {
  public static void main(String args[])
    throws java.io.IOException {
    char choice;

    for(;;) {
      do {
        System.out.println("Help on:");
        System.out.println("  1. if");
        System.out.println("  2. switch");
        System.out.println("  3. for");
        System.out.println("  4. while");
        System.out.println("  5. do-while");
        System.out.println("  6. break");
        System.out.println("  7. continue\n");
        System.out.print("Choose one (q to quit): ");
        do {
          choice = (char) System.in.read();
        } while(choice == '\n' | choice == '\r');
      } while( choice < '1' | choice > '7' & choice != 'q');

      if(choice == 'q') break;

      System.out.println("\n");

      switch(choice) {
        case '1':
          System.out.println("The if:\n");
          System.out.println("if(condition) statement;");
          System.out.println("else statement;");
          break;
        case '2':
          System.out.println("The switch:\n");
          System.out.println("switch(expression) {");
          System.out.println("  case constant:");
          System.out.println("    statement sequence");
          System.out.println("    break;");
          System.out.println("  // ...");
          System.out.println("}");
          break;
```

```
          case '3':
            System.out.println("The for:\n");
            System.out.print("for(init; condition; iteration)");
            System.out.println(" statement;");
            break;
          case '4':
            System.out.println("The while:\n");
            System.out.println("while(condition) statement;");
            break;
          case '5':
            System.out.println("The do-while:\n");
            System.out.println("do {");
            System.out.println("  statement;");
            System.out.println("} while (condition);");
            break;
          case '6':
            System.out.println("The break:\n");
            System.out.println("break; or break label;");
            break;
          case '7':
            System.out.println("The continue:\n");
            System.out.println("continue; or continue label;");
            break;
        }
        System.out.println();
      }
    }
  }
```

6. Here is a sample run:

```
Help on:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Choose one (q to quit): 1

The if:

if(condition) statement;
else statement;

Help on:
```

(continued)

```
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Choose one (q to quit): 6

The break:

break; or break label;

Help on:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Choose one (q to quit): q
```

CRITICAL SKILL 3.10

Nested Loops

As you have seen in some of the preceding examples, one loop can be nested inside of another. Nested loops are used to solve a wide variety of programming problems and are an essential part of programming. So, before leaving the topic of Java's loop statements, let's look at one more nested loop example. The following program uses a nested **for** loop to find the factors of the numbers from 2 to 100.

```
/*
   Use nested loops to find factors of numbers
   between 2 and 100.
*/
class FindFac {
  public static void main(String args[]) {

    for(int i=2; i <= 100; i++) {
      System.out.print("Factors of " + i + ": ");
      for(int j = 2; j < i; j++)
        if((i%j) == 0) System.out.print(j + " ");
      System.out.println();
    }
```

```
  }
}
```

Here is a portion of the output produced by the program:

```
Factors of 2:
Factors of 3:
Factors of 4: 2
Factors of 5:
Factors of 6: 2 3
Factors of 7:
Factors of 8: 2 4
Factors of 9: 3
Factors of 10: 2 5
Factors of 11:
Factors of 12: 2 3 4 6
Factors of 13:
Factors of 14: 2 7
Factors of 15: 3 5
Factors of 16: 2 4 8
Factors of 17:
Factors of 18: 2 3 6 9
Factors of 19:
Factors of 20: 2 4 5 10
```

In the program, the outer loop runs **i** from 2 through 100. The inner loop successively tests all numbers from 2 up to **i**, printing those that evenly divide **i**. Extra challenge: The preceding program can be made more efficient. Can you see how? (Hint: the number of iterations in the inner loop can be reduced.)

✓ Module 3 Mastery Check

1. Write a program that reads characters from the keyboard until a period is received. Have the program count the number of spaces. Report the total at the end of the program.
2. Show the general form of the **if-else-if** ladder.
3. Given

```
if(x < 10)
  if(y > 100) {
    if(!done) x = z;
    else y = z;
  }
else System.out.println("error"); // what if?
```

to what **if** does the last **else** associate?