

# 5

## Control Structures: Part 2

### Objectives

- To be able to use the **for** and **do/while** repetition structures to execute statements in a program repeatedly.
- To understand multiple selection using the **switch** selection structure.
- To be able to use the **break** and **continue** program control statements.
- To be able to use the logical operators.

*Who can control his fate?*

William Shakespeare, *Othello*

*The used key is always bright.*

Benjamin Franklin

*Man is a tool-making animal.*

Benjamin Franklin

*Intelligence ... is the faculty of making artificial objects,  
especially tools to make tools.*

Henri Bergson



## Outline

- 5.1 Introduction
- 5.2 Essentials of Counter-Controlled Repetition
- 5.3 The `for` Repetition Structure
- 5.4 Examples Using the `for` Structure
- 5.5 The `switch` Multiple-Selection Structure
- 5.6 The `do/while` Repetition Structure
- 5.7 Statements `break` and `continue`
- 5.8 Labeled `break` and `continue` Statements
- 5.9 Logical Operators
- 5.10 Structured Programming Summary
- 5.11 (Optional Case Study) Thinking About Objects: Identifying Objects' States and Activities

*Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises*

## 5.1 Introduction

Chapter 4 began our introduction to the types of building blocks that are available for problem solving and used those building blocks to employ proven program construction principles. In this chapter, we continue our presentation of the theory and principles of structured programming by introducing Java's remaining control structures. As in Chapter 4, the Java techniques you learn here are applicable to most high-level languages. When we begin our formal treatment of object-based programming in Java in Chapter 8, we will see that the control structures we study in this chapter and Chapter 4 are helpful in building and manipulating objects.

## 5.2 Essentials of Counter-Controlled Repetition

Counter-controlled repetition requires the following:

1. the *name* of a control variable (or loop counter),
2. the *initial value* of the control variable,
3. the amount of *increment* (or *decrement*) by which the control variable is modified each time through the loop (also known as *each iteration of the loop*), and
4. the condition that tests for the *final value* of the control variable (i.e., whether looping should continue).

To see the four elements of counter-controlled repetition, consider the applet shown in Fig. 5.1, which draws 10 lines from the applet's `paint` method. Remember that an applet requires a separate HTML document to load the applet into the `appletviewer` or a browser. For the purpose of this applet, the `<applet>` tag specifies a width of **275** pixels and a height of **110** pixels.

```

1  // Fig. 5.1: WhileCounter.java
2  // Counter-controlled repetition
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.JApplet;
9
10 public class WhileCounter extends JApplet {
11
12     // draw lines on applet's background
13     public void paint( Graphics g )
14     {
15         // call inherited version of method paint
16         super.paint( g );
17
18         int counter = 1;           // initialization
19
20         while ( counter <= 10 ) {  // repetition condition
21             g.drawLine( 10, 10, 250, counter * 10 );
22             ++counter;             // increment
23         } // end while structure
24     } // end method paint
25
26 } // end class WhileCounter

```

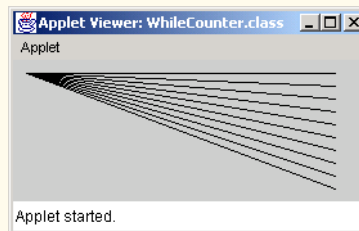


Fig. 5.1 Counter-controlled repetition.

The applet's **paint** method (that the applet container calls when it executes the applet) operates as follows: The declaration on line 18 *names* the control variable (**counter**), declares it to be an integer, reserves space for it in memory and sets it to an *initial value* of **1**. Declarations that include initialization are, in effect, executable statements. The declaration and initialization of **counter** could also have been accomplished with the declaration and statement

```

int counter; // declare counter
counter = 1; // initialize counter to 1

```

The declaration is not executable, but the assignment statement is. We use both methods of initializing variables throughout this book.

Line 21 in the **while** structure uses **Graphics** reference **g**, which refers to the applet's **Graphics** object, to send the **drawLine** message to the **Graphics** object, asking it to draw a line. Remember that “sending a message to an object” actually means calling a method to perform a task. One of the **Graphics** object's many services is to draw lines. In previous chapters, we also saw that the **Graphics** object's other services include drawing rectangles, strings and ovals. **Graphics** method **drawLine** requires four arguments, representing the line's first *x*-coordinate, first *y*-coordinate, second *x*-coordinate and second *y*-coordinate. In this example, the second *y*-coordinate changes value during each iteration of the loop with the calculation **counter \* 10**. This change causes the second point (the end point of the line) in each call to **drawLine** to move 10 pixels down the applet's display area.

Line 22 in the **while** structure *increments* the control variable by 1 for each iteration of the loop. The loop-continuation condition in the **while** structure tests whether the value of the control variable is less than or equal to **10** (the *final value* for which the condition is **true**). Note that the program performs the body of this **while** structure even when the control variable is **10**. The loop terminates when the control variable exceeds **10** (i.e., **counter** becomes **11**).

The program in Fig. 5.1 can be made more concise by initializing **counter** to **0** and preincrementing **counter** in the **while** structure condition as follows:

```
while ( ++counter <= 10 )    // repetition condition
    g.drawLine( 10, 10, 250, counter * 10 );
```

This code saves a statement (and eliminates the need for braces around the loop's body), because the **while** condition performs the increment before testing the condition. (Remember that the precedence of **++** is higher than that of **<=**.) Coding in such a condensed fashion takes practice.



### Good Programming Practice 5.1

*Programs should control counting loops with integer values.*



### Common Programming Error 5.1

*Because floating-point values may be approximate, controlling the counting of loops with floating-point variables may result in imprecise counter values and inaccurate tests for termination.*



### Good Programming Practice 5.2

*Indent the statements in the body of each control structure.*



### Good Programming Practice 5.3

*Put a blank line before and after each major control structure to make it stand out in the program.*



### Good Programming Practice 5.4

*Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of nesting.*



### Good Programming Practice 5.5

*Vertical spacing above and below control structures, and indentation of the bodies of control structures within the control structures' headers, gives programs a two-dimensional appearance that enhances readability.*

## 5.3 The for Repetition Structure

The **for** repetition structure handles all of the details of counter-controlled repetition. To illustrate the power of the **for** structure, let us rewrite the applet of Fig. 5.1. The result is shown in Fig. 5.2. Remember that this program requires a separate HTML document to load the applet into the **appletviewer**. For the purpose of this applet, the **<applet>** tag specifies a width of **275** pixels and a height of **110** pixels.

```

1  // Fig. 5.2: ForCounter.java
2  // Counter-controlled repetition with the for structure
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.JApplet;
9
10 public class ForCounter extends JApplet {
11
12     // draw lines on applet's background
13     public void paint( Graphics g )
14     {
15         // call inherited version of method paint
16         super.paint( g );
17
18         // Initialization, repetition condition and incrementing
19         // are all included in the for structure header.
20         for ( int counter = 1; counter <= 10; counter++ )
21             g.drawLine( 10, 10, 250, counter * 10 );
22
23     } // end method paint
24
25 } // end class ForCounter

```

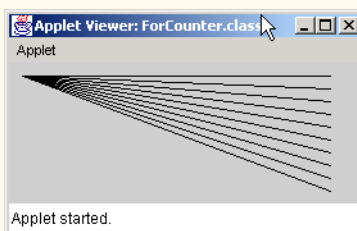


Fig. 5.2 Counter-controlled repetition with the **for** structure.

The applet's **paint** method operates as follows: When the **for** structure (lines 20–21) begins executing, the control variable **counter** is initialized to **1**. (The first two elements of counter-controlled repetition and the *name* of the control variable and its *initial value*.) Next, the program checks the loop-continuation condition, **counter <= 10**. The condition contains the *final value* (**10**) of the control variable. Because the initial value of **counter** is **1**, the condition is satisfied (**true**), so the body statement (line 21) draws a line. After executing the body of the loop, the program increments variable **counter** in the expression **counter++**. Then, the program performs the loop-continuation test again to determine whether the program should continue with the next iteration of the loop or whether it should terminate the loop. At this point, the control variable value is **2**, so the condition is true (i.e., the final value is not exceeded), and thus the program performs the body statement again (i.e., the next iteration of the loop). This process continues until the **counter**'s value becomes **11**, causing the loop-continuation test to fail and repetition to terminate. Then, the program performs the first statement after the **for** structure. (In this case, method **paint** terminates, because the program reaches the end of **paint**.)

Notice that Fig. 5.2 uses the loop-continuation condition **counter <= 10**. If the programmer incorrectly specified **counter < 10** as the condition, the loop would be executed only nine times. This mistake is a common logic error called an *off-by-one error*.



### Common Programming Error 5.2

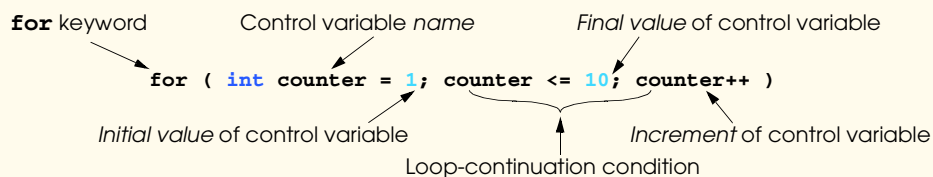
Using an incorrect relational operator or using an incorrect final value of a loop counter in the condition of a **while**, **for** or **do/while** structure can cause an *off-by-one error*.



### Good Programming Practice 5.6

Using the final value in the condition of a **while** or **for** structure and using the **<=** relational operator will help avoid off-by-one errors. For a loop that prints the values 1 to 10, the loop-continuation condition should be **counter <= 10** rather than **counter < 10** (which causes an off-by-one error) or **counter < 11** (which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times, **counter** would be initialized to zero and the loop-continuation test would be **counter < 10**.

Figure 5.3 takes a closer look at the **for** structure of Fig. 5.2. The **for** structure's first line (including the keyword **for** and everything in parentheses after **for**) is sometimes called the **for structure header**. Notice that the **for** structure “does it all”: It specifies each of the items needed for counter-controlled repetition with a control variable. If there is more than one statement in the body of the **for** structures, braces (**{** and **}**) are required to define the body of the loop.



**Fig. 5.3** Components of a typical **for** structure header.

The general format of the **for** structure is

```
for ( expression1; expression2; expression3 )
    statement
```

where *expression1* names the loop's control variable and provides its initial value, *expression2* is the loop-continuation condition (containing the control variable's final value) and *expression3* modifies the value of the control variable, so that the loop-continuation condition eventually becomes **false**. In most cases, the **for** structure can be represented with an equivalent **while** structure, with *expression1*, *expression2* and *expression3* placed as follows:

```
expression1;
while ( expression2 ) {
    statement
    expression3;
}
```

In Section 5.7, we show a case in which a **for** structure cannot be represented with an equivalent **while** structure.

If *expression1* (the initialization section) declares the control variable inside the parentheses of the header of the **for** structure (i.e., the control variable's type is specified before the name of the variable), the control variable can be used only in the **for** structure. This restricted use of the name of the control variable is known as the variable's *scope*. The scope of a variable defines where the program can use the variable. For example, we mentioned previously that a program can use a local variable only in the method that declares the variable. Scope is discussed in detail in Chapter 6, "Methods."

### Common Programming Error 5.3



When the control variable of a **for** structure is initially defined in the initialization section of the header of the **for** structure, using the control variable after the body of the structure is a syntax error.

Sometimes, *expression1* and *expression3* in a **for** structure are comma-separated lists of expressions that enable the programmer to use multiple initialization expressions and/or multiple increment expressions. For example, there may be several control variables in a single **for** structure that must be initialized and incremented.

### Good Programming Practice 5.7



Place only expressions involving the control variables in the initialization and increment sections of a **for** structure. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the body of the loop (if they execute once per iteration of the loop, like incrementing or decrementing statements).

The three expressions in the **for** structure are optional. If *expression2* is omitted, Java assumes that the loop-continuation condition is **true**, thus creating an infinite loop. One might omit *expression1* if the program initializes the control variable before the loop. One might omit *expression3* if the program calculates the increment with statements in the loop's body or if the loop does not require an increment. The increment expression in the **for** structure acts as a stand-alone statement at the end of the body of the **for** structure, so the expressions

```

counter = counter + 1
counter += 1
++counter
counter++

```

are equivalent in the increment portion of the **for** structure. Many programmers prefer the form **counter++**, because a **for** structure increments its control variable after the body of the loop executes and placing **++** after the variable name increments the variable after the program uses its value. Therefore, the postincrementing form seems more natural. Pre-incrementing and postincrementing *have the same effect in the increment expression*, because the increment does not appear in a larger expression. The two semicolons in the **for** structure are required.



#### Common Programming Error 5.4

Using commas instead of the two required semicolons in a **for** header is a syntax error.



#### Common Programming Error 5.5

Placing a semicolon immediately to the right of the right parenthesis of a **for** header makes the body of that **for** structure an empty statement. This is normally a logic error.

The initialization, loop-continuation condition and increment portions of a **for** structure can contain arithmetic expressions. For example, assume that **x = 2** and **y = 10**. If **x** and **y** are not modified in the body of the loop, the statement

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to the statement

```
for ( int j = 2; j <= 80; j += 5 )
```

The increment of a **for** structure may also be negative, in which case it is really a decrement, and the loop actually counts downward.

If the loop-continuation condition is initially **false**, the program does not perform the body of the **for** structure. Instead, execution proceeds with the statement following the **for** structure.

Programs frequently display the control variable value or use it in calculations in loop body. However, this use is not required. It is common to use the control variable for controlling repetition while never mentioning it in the body of the **for** structure.



#### Testing and Debugging Tip 5.1

Although the value of the control variable can be changed in the body of a **for** loop, avoid doing so, because this practice can lead to subtle errors.

We flowchart the **for** structure much as we do the **while** structure. For example, the flowchart of the **for** statement

```

for ( int counter = 1; counter <= 10; counter++ )
    g.drawLine( 10, 10, 250, counter * 10 );

```

is shown in Fig. 5.4. This flowchart makes it clear that the initialization occurs only once and that the increment occurs each time *after* the program performs the body statement.

Note that, besides small circles and arrows, the flowchart contains only rectangle symbols and a diamond symbol. The programmer fills the rectangles and diamonds with actions and decisions appropriate to the algorithm.

## 5.4 Examples Using the **for** Structure

The examples given next show methods of varying the control variable in a **for** structure. In each case, we write the appropriate **for** structure header. Note the change in the relational operator for loops that decrement the control variable.

- a) Vary the control variable from **1** to **100** in increments of **1**.

```
for ( int i = 1; i <= 100; i++ )
```

- b) Vary the control variable from **100** to **1** in increments of **-1** (i.e., decrements of **1**).

```
for ( int i = 100; i >= 1; i-- )
```

- c) Vary the control variable from **7** to **77** in steps of **7**.

```
for ( int i = 7; i <= 77; i += 7 )
```

- d) Vary the control variable from **20** to **2** in steps of **-2**.

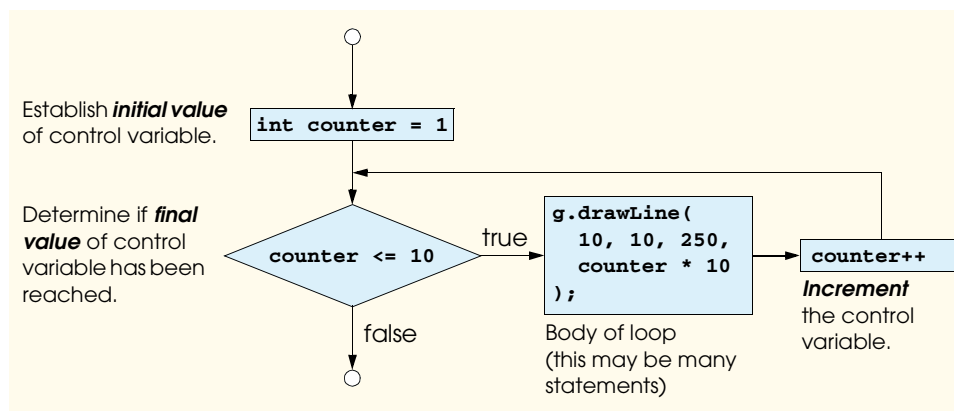
```
for ( int i = 20; i >= 2; i -= 2 )
```

- e) Vary the control variable over the following sequence of values: **2, 5, 8, 11, 14, 17, 20**.

```
for ( int j = 2; j <= 20; j += 3 )
```

- f) Vary the control variable over the following sequence of values: **99, 88, 77, 66, 55, 44, 33, 22, 11, 0**.

```
for ( int j = 99; j >= 0; j -= 11 )
```



**Fig. 5.4** Flowcharting a typical **for** repetition structure.



### Common Programming Error 5.6

*Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (such as using `i <= 1` in a loop counting down to 1) is usually a logic error and will yield incorrect results when the program runs.*

The next two sample programs demonstrate simple applications of the **for** repetition structure. The application in Fig. 5.5 uses the **for** structure to sum all the even integers from 2 to 100. Remember that the **java** interpreter is used to execute an application from the command window.

Note that the body of the **for** structure in Fig. 5.5 could actually be merged into the rightmost portion of the **for** header by using a comma as follows:

```
for ( int number = 2;
      number <= 100;
      sum += number, number += 2 )
; // empty statement
```

```
1 // Fig. 5.5: Sum.java
2 // Counter-controlled repetition with the for structure
3
4 // Java extension packages
5 import javax.swing.JOptionPane;
6
7 public class Sum {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12        int sum = 0;
13
14        // sum even integers from 2 through 100
15        for ( int number = 2; number <= 100; number += 2 )
16            sum += number;
17
18        // display results
19        JOptionPane.showMessageDialog( null, "The sum is " + sum,
20            "Sum Even Integers from 2 to 100",
21            JOptionPane.INFORMATION_MESSAGE );
22
23        System.exit( 0 ); // terminate the application
24
25    } // end method main
26
27 } // end class Sum
```

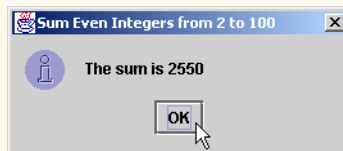


Fig. 5.5 Summation with the **for** structure.

Similarly, the initialization `sum = 0` could be merged into the initialization section of the `for` structure.



### Good Programming Practice 5.8

Although statements preceding a `for` structure and statements in the body of a `for` structure can often be merged into the header of the `for` structure, avoid doing so, because it makes the program more difficult to read.



### Good Programming Practice 5.9

Limit the size of control structure headers to a single line, if possible.

The next example uses the `for` structure to compute compound interest. Consider the following problem:

*A person invests \$1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:*

$$a = p (1 + r)^n$$

where

$p$  is the original amount invested (i.e., the principal)

$r$  is the annual interest rate

$n$  is the number of years

$a$  is the amount on deposit at the end of the  $n$ th year.

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. The solution is the application shown in Fig. 5.6.

```

1  // Fig. 5.6: Interest.java
2  // Calculating compound interest
3
4  // Java core packages
5  import java.text.NumberFormat;
6  import java.util.Locale;
7
8  // Java extension packages
9  import javax.swing.JOptionPane;
10 import javax.swing.JTextArea;
11
12 public class Interest {
13
14     // main method begins execution of Java application
15     public static void main( String args[] )
16     {
17         double amount, principal = 1000.0, rate = 0.05;
18
19         // create DecimalFormat to format floating-point numbers
20         // with two digits to the right of the decimal point
21         NumberFormat moneyFormat =
22             NumberFormat.getCurrencyInstance( Locale.US );
23     }

```

Fig. 5.6 Calculating compound interest with the `for` structure (part 1 of 2).

```

24      // create JTextArea to display output
25      JTextArea outputTextArea = new JTextArea();
26
27      // set first line of text in outputTextArea
28      outputTextArea.setText( "Year\tAmount on deposit\n" );
29
30      // calculate amount on deposit for each of ten years
31      for ( int year = 1; year <= 10; year++ ) {
32
33          // calculate new amount for specified year
34          amount = principal * Math.pow( 1.0 + rate, year );
35
36          // append one line of text to outputTextArea
37          outputTextArea.append( year + "\t" +
38                                moneyFormat.format( amount ) + "\n" );
39
40      } // end for structure
41
42      // display results
43      JOptionPane.showMessageDialog( null, outputTextArea,
44                                     "Compound Interest", JOptionPane.INFORMATION_MESSAGE );
45
46      System.exit( 0 ); // terminate the application
47
48  } // end method main
49
50 } // end class Interest

```

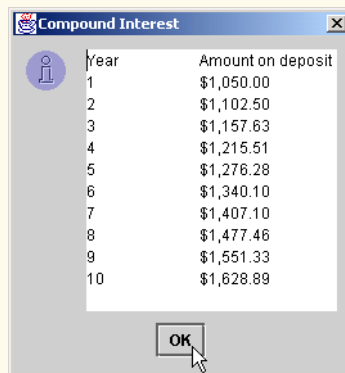


Fig. 5.6 Calculating compound interest with the **for** structure (part 2 of 2).

Line 17 in method **main** declares three **double** variables and initializes two of them—**principal** to **1000.0** and **rate** to **.05**. Java treats floating-point constants, like **1000.0** and **.05** in Fig. 5.6, as type **double**. Similarly, Java treats whole number constants, like **7** and **-22**, as type **int**. Lines 21–22 declare **NumberFormat** reference **moneyFormat** and initialize it by calling **static** method **getCurrencyInstance** of class **NumberFormat**. This method returns a **NumberFormat** object that can format numeric values as currency (e.g., in the United States, currency values normally are preceded with a dollar sign, \$). The argument to the method—**Locale.US**—indicates that

the currency values should be displayed starting with a dollar sign (\$), use a decimal point to separate dollars and cents and use a comma to delineate thousands (e.g., \$1,234.56). Class **Locale** provides constants that can be used to customize this program to represent currency values for other countries, so that currency formats are displayed properly for each *locale* (i.e., each country's local-currency format). Class **NumberFormat** (imported at line 5) is located in package **java.text**, and class **Locale** (imported at line 6) is located in package **java.util**.

Line 25 declares **JTextArea** reference **outputTextArea** and initializes it with a new object of class **JTextArea** (from package **javax.swing**). A **JTextArea** is a GUI component that can display many lines of text. The message dialog that displays the **JTextArea** determines the width and height of the **JTextArea**, based on the **String** it contains. We introduce this GUI component now because we will see many examples throughout the text in which the program outputs contain too many lines to display on the screen. This GUI component allows us to scroll through the lines of text so we can see all the program output. The methods for placing text in a **JTextArea** include **setText** and **append**.

Line 28 uses **JTextArea** method **setText** to place a **String** in the **JTextArea** to which **outputTextArea** refers. Initially, a **JTextArea** contains an empty **String** (i.e., a **String** with no characters in it). The preceding statement replaces the empty **String** with one containing the column heads for our two columns of output—"Year" and "Amount on Deposit." The column heads are separated with a tab character (escape sequence **\t**). Also, the string contains the newline character (escape sequence **\n**), indicating that any additional text appended to the **JTextArea** should begin on the next line.

The **for** structure (lines 31–40) executes its body 10 times, varying control variable **year** from 1 to 10 in increments of 1. (Note that **year** represents *n* in the statement of the problem.) Java does not include an exponentiation operator. Instead, we use **static** method **pow** of class **Math** for this purpose. **Math.pow(x, y)** calculates the value of **x** raised to the **y**th power. Method **pow** takes two arguments of type **double** and returns a **double** value. Line 34 performs the calculation from the statement of the problem,

$$a = p(1 + r)^n$$

where *a* is **amount**, *p* is **principal**, *r* is **rate** and *n* is **year**.

Lines 37–38 **append** more text to the end of the **outputTextArea**. The text includes the current value of **year**, a tab character (to position to the second column), the result of the method call **moneyFormat.format( amount )**—which formats the **amount** as U. S. currency—and a newline character (to position the cursor in the **JTextArea** at the beginning of the next line).

Lines 43–44 display the results in a message dialog. Until now, the message displayed has always been a **String**. In this example, the second argument is **outputTextArea**—a GUI component. An interesting feature of class **JOptionPane** is that the message it displays with **showMessageDialog** can be a **String** or a GUI component, such as a **JTextArea**. In this example, the message dialog sizes itself to accommodate the **JTextArea**. We use this technique several times early in this chapter to display large text-based outputs. Later in this chapter, we demonstrate how to add a scrolling capability to the **JTextArea**, so the user can view a program's output that is too large to display in full on the screen.

Notice that the variables **amount**, **principal** and **rate** are of type **double**. We did this for simplicity, because we are dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, this setting can cause trouble. Here is a simple explanation of what can go wrong when using **float** or **double** to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point): Two **double** dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your printout could appear as

```

  14.23
+ 18.67
-----
 32.91

```

but a person adding the individual numbers as printed would expect the sum to be 32.90. You have been warned!



### Good Programming Practice 5.10

*Do not use variables of type **float** or **double** to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. In the exercises, we explore the use of integers to perform monetary calculations. [Note: Some third-party vendors provide for-sale class libraries that perform precise monetary calculations.]*

Note that the body of the **for** structure contains the calculation **1.0 + rate**, which appears as an argument to the **Math.pow** method. In fact, this calculation produces the same result each time through the loop, so repeating the calculation every iteration of the loop is wasteful.



### Performance Tip 5.1

*Avoid placing expressions whose values do not change inside loops. But even if you do, many of today's sophisticated optimizing compilers will place such expressions outside loops in the generated compiled code.*



### Performance Tip 5.2

*Many compilers contain optimization features that improve the code that you write, but it is still better to write good code from the start.*

## 5.5 The **switch** Multiple-Selection Structure

We have discussed the **if** single-selection structure and the **if/else** double-selection structure. Occasionally, an algorithm contains a series of decisions in which the algorithm tests a variable or expression separately for each of the constant integral values (i.e., values of types **byte**, **short**, **int** and **char**) the variable or expression may assume and takes different actions based on those values. Java provides the **switch** multiple-selection structure to handle such decision making. The applet of Fig. 5.7 demonstrates drawing lines, rectangles or ovals, based on an integer the user inputs via an input dialog.

```

1  // Fig. 5.7: SwitchTest.java
2  // Drawing lines, rectangles or ovals based on user input.
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class SwitchTest extends JApplet {
11     int choice;    // user's choice of which shape to draw
12
13     // initialize applet by obtaining user's choice
14     public void init()
15     {
16         String input; // user's input
17
18         // obtain user's choice
19         input = JOptionPane.showInputDialog(
20             "Enter 1 to draw lines\n" +
21             "Enter 2 to draw rectangles\n" +
22             "Enter 3 to draw ovals\n" );
23
24         // convert user's input to an int
25         choice = Integer.parseInt( input );
26     }
27
28     // draw shapes on applet's background
29     public void paint( Graphics g )
30     {
31         // call inherited version of method paint
32         super.paint( g );
33
34         // loop 10 times, counting from 0 through 9
35         for ( int i = 0; i < 10; i++ ) {
36
37             // determine shape to draw based on user's choice
38             switch ( choice ) {
39
40                 case 1:
41                     g.drawLine( 10, 10, 250, 10 + i * 10 );
42                     break; // done processing case
43
44                 case 2:
45                     g.drawRect( 10 + i * 10, 10 + i * 10,
46                               50 + i * 10, 50 + i * 10 );
47                     break; // done processing case
48
49                 case 3:
50                     g.drawOval( 10 + i * 10, 10 + i * 10,
51                               50 + i * 10, 50 + i * 10 );
52                     break; // done processing case
53

```

Fig. 5.7 An example using **switch** (part 1 of 3).

```
54         default:
55             g.drawString( "Invalid value entered",
56                 10, 20 + i * 15 );
57
58     } // end switch structure
59
60 } // end for structure
61
62 } // end paint method
63
64 } // end class SwitchTest
```

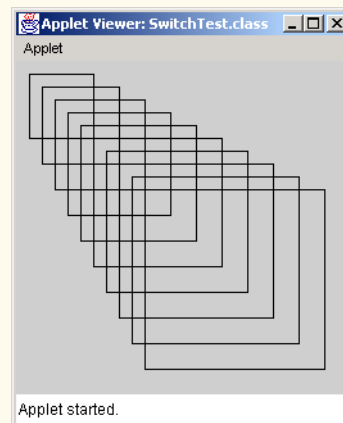
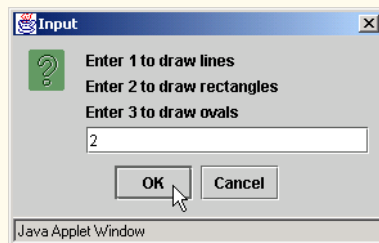
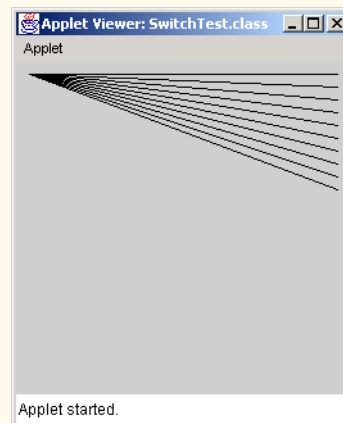
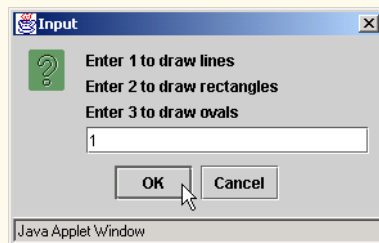


Fig. 5.7 An example using **switch** (part 2 of 3).

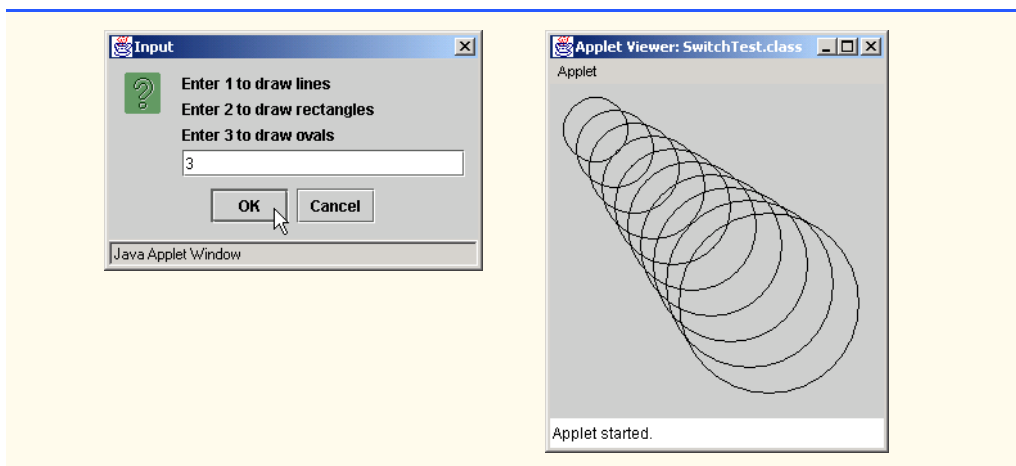


Fig. 5.7 An example using **switch** (part 3 of 3).

Line 11 in applet **SwitchTest** defines instance variable **choice** of type **int**. This variable stores the user's input that determines which type of shape to draw in **paint**.

Method **init** (lines 14–26) declares local variable **input** of type **String** in line 16. This variable stores the **String** the user types in the input dialog. Lines 19–22 display the input dialog with **static** method **JOptionPane.showInputDialog** and prompt the user to enter **1** to draw lines, **2** to draw rectangles or **3** to draw ovals. Line 25 converts input from a **String** to an **int** and assigns the result to **choice**.

Method **paint** (lines 29–62) contains a **for** structure (lines 35–60) that loops 10 times. In this example, the **for** structure's header, in line 35, uses zero-based counting. The values of **i** for the 10 iterations of the loop are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, and the loop terminates when **i**'s value becomes 10. [Note: As you know, the applet container calls method **paint** after methods **init** and **start**. The applet container also calls method **paint** whenever the applet's screen area must be refreshed—e.g., after another window that covered the applet's area is moved to a different location on the screen.]

Nested in the **for** structure's body is a **switch** structure (lines 38–58) that draws shapes based on the integer value input by the user in method **init**. The **switch** structure consists of a series of **case labels** and an optional **default case**.

When the flow of control reaches the **switch** structure, the program evaluates the *controlling expression* (**choice**) in the parentheses following keyword **switch**. The program compares the value of the controlling expression (which must evaluate to an integral value of type **byte**, **char**, **short** or **int**) with each **case label**. Assume that the user entered the integer **2** as his or her choice. The program compares **2** with each **case** in the **switch**. If a match occurs (**case 2:**), the program executes the statements for that **case**. For the integer **2**, lines 44–47 draw a rectangle, using four arguments, representing the upper left *x*-coordinate, upper left *y*-coordinate, width and height of the rectangle, and the **switch** structure exits immediately with the **break** statement. Then, the program increments the counter variable in the **for** structure and reevaluates the loop-continuation condition to determine whether to perform another iteration of the loop.

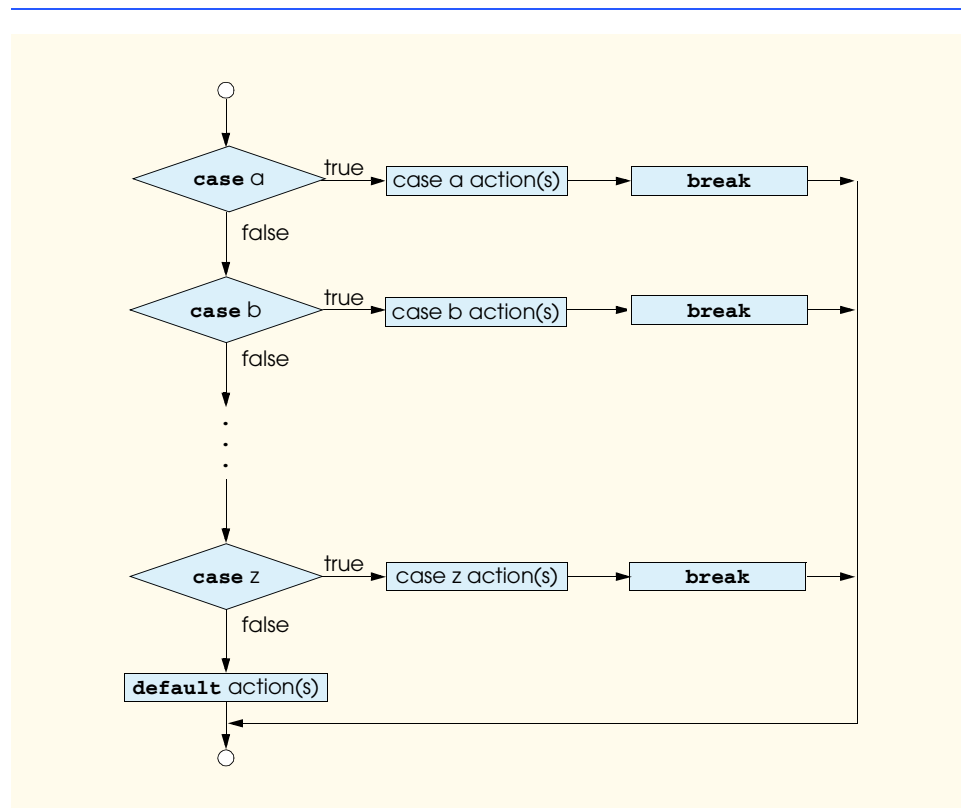
The **break** statement causes program control to proceed with the first statement after the **switch** structure. (In this case, we reach the end of the **for** structure's body, so con-

control flows to the control variable's increment expression in the header of the **for** structure.) Without **break**, the **cases** in a **switch** statement would run together. Each time a match occurs in the structure, the statements for all the remaining **cases** will execute. (This feature is perfect for programming the iterative song "The Twelve Days of Christmas.") If no match occurs between the controlling expression's value and a **case** label, the **default** case executes, and the program draws an error message on the applet.

Each **case** can have multiple actions. The **switch** structure differs from other structures in that it does not require braces around multiple actions in each **case**. Figure 5.8 shows the general **switch** structure flowchart (using a **break** in each **case**). [Note: As an exercise, make a flowchart of the general **switch** structure without **breaks**.]

The flowchart makes it clear that each **break** statement at the end of a **case** causes control to exit the **switch** structure immediately. The **break** statement is not required for the last **case** in the **switch** structure (or the **default** case, when it appears last), because the program continues with the next statement after the **switch**.

Again, note that, besides small circles and arrows, the flowchart contains only rectangle and diamond symbols. It is the programmer's responsibility to fill the rectangles and diamonds with actions and decisions appropriate to the algorithm. Although nested control structures are common, it is rare to find nested **switch** structures in a program.



**Fig. 5.8** The **switch** multiple-selection structure.



### Common Programming Error 5.7

Forgetting a **break** statement when one is needed in a **switch** structure is a logic error.



### Good Programming Practice 5.11

Provide a **default** case in **switch** statements. Cases not explicitly tested in a **switch** statement without a **default** case are ignored. Including a **default** case focuses the programmer on the need to process exceptional conditions. There are situations in which no **default** processing is needed.



### Good Programming Practice 5.12

Although the **cases** and the **default** case in a **switch** structure can occur in any order, it is considered a good programming practice to place the **default** clause last.



### Good Programming Practice 5.13

In a **switch** structure, when the **default** clause is listed last, the **break** for that **case** statement is not required. Some programmers include this **break** for clarity and symmetry with other cases.

Note that listing **case** labels together (such as **case 1: case 2:** with no statements between the cases) performs the same set of actions for each case.

When using the **switch** structure, remember that the expression after each **case** can be only a *constant integral expression* (i.e., any combination of character constants and integer constants that evaluates to a constant integer value). A character constant is represented as the specific character in single quotes, such as **'A'**. An integer constant is simply an integer value. The expression after each **case** also can be a *constant variable*—i.e., a variable that contains a value which does not change for the entire program. Such a variable is declared with keyword **final** (discussed in Chapter 6). When we discuss object-oriented programming in Chapter 9, we present a more elegant way to implement **switch** logic. We use a technique called *polymorphism* to create programs that are often clearer, easier to maintain and easier to extend than programs using **switch** logic.

## 5.6 The **do/while** Repetition Structure

The **do/while** repetition structure is similar to the **while** structure. In the **while** structure, the program tests the loop-continuation condition at the beginning of the loop, before performing the body of the loop. The **do/while** structure tests the loop-continuation condition *after* performing the body of the loop; therefore, *the loop body always executes at least once*. When a **do/while** structure terminates, execution continues with the statement after the **while** clause. Note that it is not necessary to use braces in the **do/while** structure if there is only one statement in the body. However, most programmers include the braces, to avoid confusion between the **while** and **do/while** structures. For example,

```
while ( condition )
```

normally is the first line of a **while** structure. A **do/while** structure with no braces around a single-statement body appears as

```
do
    statement
while ( condition );
```

which can be confusing. Reader may misinterpret the last line—**while**( *condition* );—as a **while** structure containing an empty statement (the semicolon by itself). Thus, to avoid confusion, the **do/while** structure with one statement often is written as follows:

```
do {
    statement
} while ( condition );
```



#### Good Programming Practice 5.14

Some programmers always include braces in a **do/while** structure, even if the braces are not necessary. This helps eliminate ambiguity between the **while** structure and the **do/while** structure containing only one statement.



#### Common Programming Error 5.8

Infinite loops occur when the loop-continuation condition in a **while**, **for** or **do/while** structure never becomes **false**. To prevent this situation, make sure that there is not a semicolon immediately after the header of a **while** or **for** structure. In a counter-controlled loop, ensure that the control variable is incremented (or decremented) in the body of the loop. In a sentinel-controlled loop, ensure that the sentinel value is eventually input.

The applet in Fig. 5.9 uses a **do/while** structure to draw 10 nested circles, using **Graphics** method **drawOval**.

```
1  // Fig. 5.9: DoWhileTest.java
2  // Using the do/while repetition structure.
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.JApplet;
9
10 public class DoWhileTest extends JApplet {
11
12     // draw lines on applet's background
13     public void paint( Graphics g )
14     {
15         // call inherited version of method paint
16         super.paint( g );
17
18         int counter = 1;
19
20         do {
21             g.drawOval( 110 - counter * 10, 110 - counter * 10,
22                         counter * 20, counter * 20 );
23             ++counter;
24         } while ( counter <= 10 ); // end do/while structure
25
26     } // end method paint
27
28 } // end class DoWhileTest
```

Fig. 5.9 Using the **do/while** repetition structure (part 1 of 2).

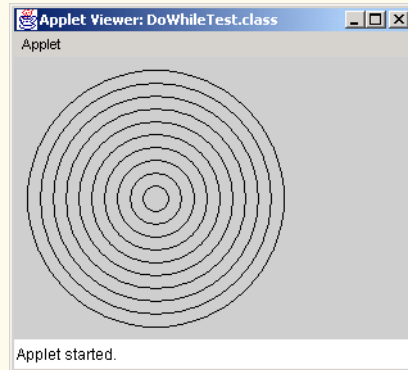


Fig. 5.9 Using the **do/while** repetition structure (part 2 of 2).

In method **paint** (lines 13–26), line 18 declares control variable **counter** and initializes it to **1**. Upon entering the **do/while** structure, lines 21–22 send the **drawOval** message to the **Graphics** object to which **g** refers. The four arguments that represent the upper left *x*-coordinate, upper left *y*-coordinate, width and height of the oval's *bounding box* (an imaginary rectangle in which the oval touches the center of all four sides of the rectangle) are calculated based on the value of **counter**. The program draws the innermost oval first. The bounding box's upper left corner for each subsequent oval moves closer to the upper left corner of the applet. At the same time, the width and height of the bounding box are increased, to ensure that each new oval contains all the previous ovals. Line 23 increments **counter**. Then, the program evaluates the loop-continuation test at the bottom of the loop. The **do/while** flowchart in Fig. 5.10 makes it clear that the program does not evaluate the loop-continuation condition until after the action executes once.

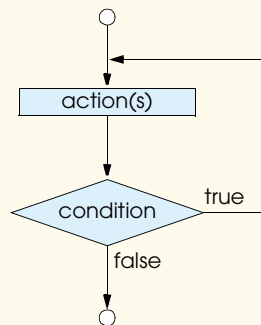


Fig. 5.10 Flowcharting the **do/while** repetition structure.

## 5.7 Statements **break** and **continue**

The **break** and **continue** statements alter the flow of control. The **break** statement, when executed in a **while**, **for**, **do/while** or **switch** structure, causes immediate exit from that structure. Execution continues with the first statement after the structure. Common uses of the **break** statement are to escape early from a loop or skip the remainder of a **switch** structure (as in Fig. 5.7). Figure 5.11 demonstrates the **break** statement in a **for** repetition structure.

```

1  // Fig. 5.11: BreakTest.java
2  // Using the break statement in a for structure
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class BreakTest {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         String output = "";
13         int count;
14
15         // loop 10 times
16         for ( count = 1; count <= 10; count++ ) {
17
18             // if count is 5, terminate loop
19             if ( count == 5 )
20                 break; // break loop only if count == 5
21
22             output += count + " ";
23
24         } // end for structure
25
26         output += "\nBroke out of loop at count = " + count;
27         JOptionPane.showMessageDialog( null, output );
28
29         System.exit( 0 ); // terminate application
30
31     } // end method main
32
33 } // end class BreakTest

```

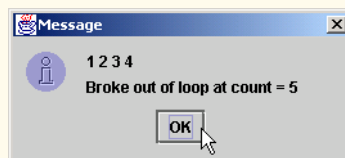


Fig. 5.11 Using the **break** statement in a **for** structure.

When the **if** structure at line 19 in the **for** structure detects that **count** is 5, the **break** statement at line 20 executes. This statement terminates the **for** structure, and the program proceeds to line 26 (immediately after the **for**). Line 26 completes the string to display in a message dialog at line 27. The loop fully executes its body only four times.

The **continue** statement, when executed in a **while**, **for** or **do/while** structure, skips the remaining statements in the loop body and proceeds with the next iteration of the loop. In **while** and **do/while** structures, the program evaluates the loop-continuation test immediately after the **continue** statement executes. In **for** structures, the increment expression executes, then the program evaluates the loop-continuation test. Earlier, we stated that the **while** structure could be used in most cases to represent the **for** structure. The one exception occurs when the increment expression in the **while** structure follows the **continue** statement. In this case, the increment does not execute before the program evaluates the repetition-continuation condition, so the **while** structure does not execute in the same manner as does the **for** structure. Figure 5.12 uses the **continue** statement in a **for** structure to skip the string concatenation statement (line 22) when the **if** structure (line 18) determines that the value of **count** is 5. When the **continue** statement executes, program control continues with the increment of the control variable in the **for** structure.



### Good Programming Practice 5.15

*Some programmers feel that **break** and **continue** violate structured programming. Because the effects of these statements are achievable with structured programming techniques, these programmers do not use **break** and **continue**.*

```

1  // Fig. 5.12: ContinueTest.java
2  // Using the continue statement in a for structure
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class ContinueTest {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         String output = "";
13
14         // loop 10 times
15         for ( int count = 1; count <= 10; count++ ) {
16
17             // if count is 5, continue with next iteration of loop
18             if ( count == 5 )
19                 continue; // skip remaining code in loop
20                             // only if count == 5
21
22             output += count + " ";
23
24         } // end for structure
25

```

Fig. 5.12 Using the **continue** statement in a **for** structure (part 1 of 2).

```

26     output += "\nUsed continue to skip printing 5";
27     JOptionPane.showMessageDialog( null, output );
28
29     System.exit( 0 ); // terminate application
30
31 } // end method main
32
33 } // end class ContinueTest

```

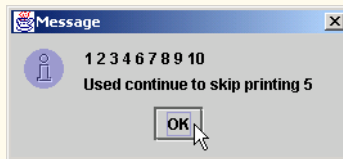


Fig. 5.12 Using the **continue** statement in a **for** structure (part 2 of 2).



### Performance Tip 5.3

The **break** and **continue** statements, when used properly, perform faster than the corresponding structured techniques.



### Software Engineering Observation 5.1

There is a tension between achieving quality software engineering and achieving the best performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

## 5.8 Labeled break and continue Statements

The **break** statement can break out of only an immediately enclosing **while**, **for**, **do/while** or **switch** structure. To break out of a nested set of structures, you can use the *labeled break statement*. This statement, when executed in a **while**, **for**, **do/while** or **switch** structure, causes immediate exit from that structure and any number of enclosing repetition structures; program execution resumes with the first statement after the enclosing *labeled block* (i.e., a set of statements enclosed in curly braces and preceded by a label). The block can be either a repetition structure (the body would be the block) or a block in which the repetition structure is the first executable code. Labeled **break** statements are commonly used to terminate nested looping structures containing **while**, **for**, **do/while** or **switch** structures. Figure 5.13 demonstrates the labeled **break** statement in a nested **for** structure.

The block (lines 14–37) begins with a *label* (an identifier followed by a colon) at line 14; here, we use the label “**stop:**.” The block is enclosed in braces at the end of line 14 and line 37 and includes the nested **for** structure (lines 17–32) and the string-concatenation statement at line 35. When the **if** structure at line 23 detects that **row** is equal to 5, the **break** statement at line 24 executes. This statement terminates both the **for** structure at line 20 and its enclosing **for** structure at line 17. The program proceeds immediately to

line 39—the first statement after the labeled block. The outer **for** structure fully executes its body only four times. Notice that the string-concatenation statement at line 35 never executes, because it is in the labeled block's body, and the outer **for** structure never completes.

```

1  // Fig. 5.13: BreakLabelTest.java
2  // Using the break statement with a label
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class BreakLabelTest {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         String output = "";
13
14         stop: {    // labeled block
15
16             // count 10 rows
17             for ( int row = 1; row <= 10; row++ ) {
18
19                 // count 5 columns
20                 for ( int column = 1; column <= 5 ; column++ ) {
21
22                     // if row is 5, jump to end of "stop" block
23                     if ( row == 5 )
24                         break stop; // jump to end of stop block
25
26                     output += "* ";
27
28                 } // end inner for structure
29
30                 output += "\n";
31
32             } // end outer for structure
33
34             // the following line is skipped
35             output += "\nLoops terminated normally";
36
37         } // end labeled block
38
39         JOptionPane.showMessageDialog(
40             null, output, "Testing break with a label",
41             JOptionPane.INFORMATION_MESSAGE );
42
43         System.exit( 0 ); // terminate application
44
45     } // end method main
46
47 } // end class BreakLabelTest

```

Fig. 5.13 Using a labeled **break** statement in a nested **for** structure (part 1 of 2).

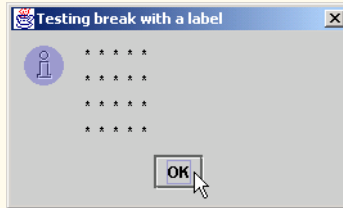


Fig. 5.13 Using a labeled **break** statement in a nested **for** structure (part 2 of 2).

The **continue** statement proceeds with the next iteration (repetition) of the immediately enclosing **while**, **for** or **do/while** structure. The *labeled continue statement*, when executed in a repetition structure (**while**, **for** or **do/while**), skips the remaining statements in that structure's body and any number of enclosing repetition structures and proceeds with the next iteration of the enclosing *labeled repetition structure* (i.e., a repetition structure preceded by a label). In labeled **while** and **do/while** structures, the program evaluates the loop-continuation test immediately after the **continue** statement executes. In a labeled **for** structure, the increment expression is executed, and then the loop-continuation test is evaluated. Figure 5.14 uses the labeled **continue** statement in a nested **for** structure to enable execution to continue with the next iteration of the outer **for** structure.

The labeled **for** structure (lines 14–32) starts at the **nextRow** label. When the **if** structure at line 24 in the inner **for** structure detects that **column** is greater than **row**, the **continue** statement at line 25 executes, and program control continues with the increment of the control variable of the outer **for** loop. Even though the inner **for** structure counts from 1 to 10, the number of \* characters output on a row never exceeds the value of **row**.



#### Performance Tip 5.4

The program in Fig. 5.14 can be made simpler and more efficient by replacing the condition in the **for** structure at line 21 with **column <= row** and removing the **if** structure at lines 24–25 from the program.

## 5.9 Logical Operators

So far, we have studied only *simple conditions*, such as **count <= 10**, **total > 1000** and **number != sentinelValue**. These conditions were expressed in terms of the relational operators **>**, **<**, **>=** and **<=** and the equality operators **==** and **!=**. Each decision tested one condition. To test multiple conditions in the process of making a decision, we performed these tests in separate statements or in nested **if** or **if/else** structures.

Java provides *logical operators* to enable programmers to form more complex conditions by combining simple conditions. The logical operators are **&&** (*logical AND*), **&** (*boolean logical AND*), **||** (*logical OR*), **|** (*boolean logical inclusive OR*), **^** (*boolean logical exclusive OR*) and **!** (*logical NOT*, also called *logical negation*).

```

1  // Fig. 5.14: ContinueLabelTest.java
2  // Using the continue statement with a label
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class ContinueLabelTest {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         String output = "";
13
14         nextRow:    // target label of continue statement
15
16         // count 5 rows
17         for ( int row = 1; row <= 5; row++ ) {
18             output += "\n";
19
20             // count 10 columns per row
21             for ( int column = 1; column <= 10; column++ ) {
22
23                 // if column greater than row, start next row
24                 if ( column > row )
25                     continue nextRow; // next iteration of
26                                     // labeled loop
27
28                 output += "* ";
29
30             } // end inner for structure
31
32         } // end outer for structure
33
34         JOptionPane.showMessageDialog(
35             null, output, "Testing continue with a label",
36             JOptionPane.INFORMATION_MESSAGE );
37
38         System.exit( 0 ); // terminate application
39
40     } // end method main
41
42 } // end class ContinueLabelTest

```

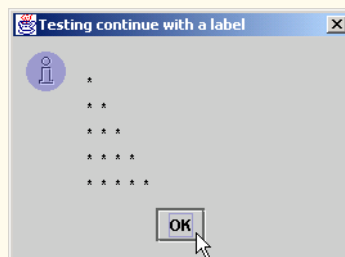


Fig. 5.14 Using a labeled **continue** statement in a nested **for** structure .

Suppose we wish to ensure at some point in a program that two conditions are *both true* before we choose a certain path of execution. In this case, we can use the logical **&&** operator, as follows:

```
if ( gender == 1 && age >= 65 )
    ++seniorFemales;
```

This **if** statement contains two simple conditions. The condition **gender == 1** might be evaluated, for example, to determine if a person is a female. The condition **age >= 65** is evaluated to determine if a person is a senior citizen. The two simple conditions are evaluated first, because the precedences of **==** and **>=** are both higher than the precedence of **&&**. The **if** statement then considers the combined condition

```
gender == 1 && age >= 65
```

This condition is **true** *if and only if* both of the simple conditions are **true**. If this combined condition is indeed **true**, the **if** structure's body statement increments variable **seniorFemales** by 1. If either or both of the simple conditions are **false**, the program skips the increment and proceeds to the statement following the **if** structure. The preceding combined condition can be made more readable by adding redundant parentheses:

```
( gender == 1 ) && ( age >= 65 )
```

The table in Fig. 5.15 summarizes the **&&** operator. The table shows all four possible combinations of **false** and **true** values for *expression1* and *expression2*. Such tables are often called *truth tables*. Java evaluates to **false** or **true** all expressions that include relational operators, equality operators and/or logical operators.

Now let us consider the **||** (logical OR) operator. Suppose we wish to ensure that either *or* both of two conditions are **true** before we choose a certain path of execution. In this case, we use the **||** operator, as in the following program segment:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
    System.out.println ( "Student grade is A" );
```

This statement also contains two simple conditions. The condition **semesterAverage >= 90** evaluates to determine if the student deserves an "A" in the course because of a solid performance throughout the semester. The condition **finalExam >= 90** evaluates to determine if the student deserves an "A" in the course because of an outstanding performance on the final exam. The **if** statement then considers the combined condition

| expression1 | expression2 | expression1 && expression2 |
|-------------|-------------|----------------------------|
| false       | false       | false                      |
| false       | true        | false                      |
| true        | false       | false                      |
| true        | true        | true                       |

Fig. 5.15 Truth table for the **&&** (logical AND) operator.

```
semesterAverage >= 90 || finalExam >= 90
```

and awards the student an “A” if either or both of the simple conditions are **true**. Note that the only time the message “**Student grade is A**” is *not* printed is when both of the simple conditions are **false**. Figure 5.16 is a truth table for the logical OR operator (`||`).

The `&&` operator has a higher precedence than the `||` operator. Both operators associate from left to right. An expression containing `&&` or `||` operators is evaluated only until truth or falsity is known. Thus, evaluation of the expression

```
gender == 1 && age >= 65
```

stops immediately if **gender** is not equal to **1** (i.e., the entire expression is **false**) and continues if **gender** is equal to **1** (i.e., the entire expression could still be **true** if the condition **age >= 65** is **true**). This performance feature for evaluation of logical AND and logical OR expressions is called *short-circuit evaluation*.



### Common Programming Error 5.9

*In expressions using operator `&&`, it is possible that a condition—we will call this the dependent condition—may require another condition to be **true** for it to be meaningful to evaluate the dependent condition. In this case, the dependent condition should be placed after the other condition, or an error might occur.*



### Performance Tip 5.5

*In expressions using operator `&&`, if the separate conditions are independent of one another, make the condition that is most likely to be **false** the leftmost condition. In expressions using operator `||`, make the condition that is most likely to be **true** the leftmost condition. This can reduce a program’s execution time.*

The *boolean logical AND* (`&`) and *boolean logical inclusive OR* (`|`) operators work identically to the regular logical AND and logical OR operators, with one exception: The boolean logical operators always evaluate both of their operands (i.e., there is no short-circuit evaluation). Therefore, the expression

```
gender == 1 & age >= 65
```

evaluates **age >= 65** regardless of whether **gender** is equal to **1**. This method is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator has a required *side effect*—a modification of a variable’s value. For example, the expression

| expression1 | expression2 | expression1    expression2 |
|-------------|-------------|----------------------------|
| false       | false       | false                      |
| false       | true        | true                       |
| true        | false       | true                       |
| true        | true        | true                       |

Fig. 5.16 Truth table for the `||` (logical OR) operator.

```
birthday == true | ++age >= 65
```

guarantees that the condition `++age >= 65` will be evaluated. Thus, the variable `age` is incremented in the preceding expression, regardless of whether the overall expression is `true` or `false`.



### Good Programming Practice 5.16

*For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they are often more trouble than they are worth.*

A condition containing the *boolean logical exclusive OR* (`^`) operator is `true` if and only if one of its operands results in a `true` value and one results in a `false` value. If both operands are `true` or both are `false`, the result of the entire condition is `false`. Figure 5.17 is a truth table for the boolean logical exclusive OR operator (`^`). This operator is also guaranteed to evaluate both of its operands (i.e., there is no short-circuit evaluation).

Java provides the `!` (logical negation) operator to enable a programmer to “reverse” the meaning of a condition. Unlike the logical operators `&&`, `&`, `||`, `|` and `^`, which combine two conditions (i.e., they are binary operators), the logical negation operator has only a single condition as an operand (i.e., they are unary operator). The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, such as in the following program segment:

```
if ( ! ( grade == sentinelValue ) )
    System.out.println( "The next grade is " + grade );
```

The parentheses around the condition `grade == sentinelValue` are needed, because the logical negation operator has a higher precedence than the equality operator. Figure 5.18 is a truth table for the logical negation operator.

In most cases, the programmer can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written as follows:

| expression1 | expression2 | expression1 ^ expression2 |
|-------------|-------------|---------------------------|
| false       | false       | false                     |
| false       | true        | true                      |
| true        | false       | true                      |
| true        | true        | false                     |

Fig. 5.17 Truth table for the boolean logical exclusive OR (`^`) operator .

| expression | !expression |
|------------|-------------|
| false      | true        |
| true       | false       |

Fig. 5.18 Truth table for operator `!` (logical negation, or logical NOT).

```

    if ( grade != sentinelValue )
        System.out.println( "The next grade is " + grade );

```

This flexibility can help a programmer express a condition in a more convenient manner.

The application in Fig. 5.19 demonstrates all of the logical operators and boolean logical operators by producing their truth tables. The program uses string concatenation to create the string that is displayed in a **JTextArea**.

```

1  // Fig. 5.19: LogicalOperators.java
2  // Demonstrating the logical operators
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class LogicalOperators {
8
9      // main method begins execution of Java application
10     public static void main( String args[] )
11     {
12         // create JTextArea to display results
13         JTextArea outputArea = new JTextArea( 17, 20 );
14
15         // attach JTextArea to a JScrollPane so user can
16         // scroll through results
17         JScrollPane scroller = new JScrollPane( outputArea );
18
19         String output;
20
21         // create truth table for && operator
22         output = "Logical AND (&&)" +
23             "\nfalse && false: " + ( false && false ) +
24             "\nfalse && true: " + ( false && true ) +
25             "\ntrue && false: " + ( true && false ) +
26             "\ntrue && true: " + ( true && true );
27
28         // create truth table for || operator
29         output += "\n\nLogical OR (||)" +
30             "\nfalse || false: " + ( false || false ) +
31             "\nfalse || true: " + ( false || true ) +
32             "\ntrue || false: " + ( true || false ) +
33             "\ntrue || true: " + ( true || true );
34
35         // create truth table for & operator
36         output += "\n\nBoolean logical AND (&)" +
37             "\nfalse & false: " + ( false & false ) +
38             "\nfalse & true: " + ( false & true ) +
39             "\ntrue & false: " + ( true & false ) +
40             "\ntrue & true: " + ( true & true );
41
42         // create truth table for | operator
43         output += "\n\nBoolean logical inclusive OR (|)" +
44             "\nfalse | false: " + ( false | false ) +
45             "\nfalse | true: " + ( false | true ) +

```

Fig. 5.19 Demonstrating the logical operators (part 1 of 2).

```

46         "\ntrue | false: " + ( true | false ) +
47         "\ntrue | true: " + ( true | true );
48
49         // create truth table for ^ operator
50         output += "\n\nBoolean logical exclusive OR (^)" +
51         "\nfalse ^ false: " + ( false ^ false ) +
52         "\nfalse ^ true: " + ( false ^ true ) +
53         "\ntrue ^ false: " + ( true ^ false ) +
54         "\ntrue ^ true: " + ( true ^ true );
55
56         // create truth table for ! operator
57         output += "\n\nLogical NOT (!)" +
58         "\n!false: " + ( !false ) +
59         "\n!true: " + ( !true );
60
61         outputArea.setText( output ); // place results in JTextArea
62
63         JOptionPane.showMessageDialog( null, scroller,
64         "Truth Tables", JOptionPane.INFORMATION_MESSAGE );
65
66         System.exit( 0 ); // terminate application
67
68     } // end method main
69
70 } // end class LogicalOperators

```

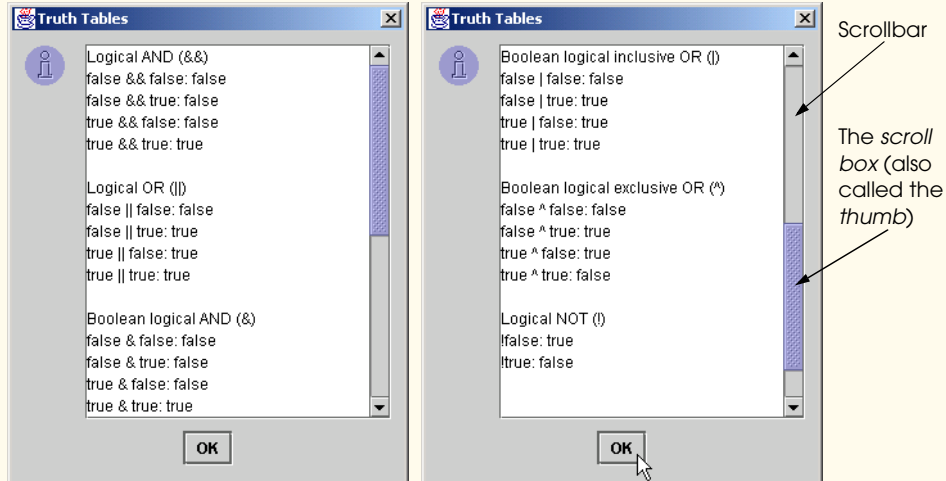


Fig. 5.19 Demonstrating the logical operators (part 2 of 2).

In the output of Fig. 5.19, the strings “true” and “false” indicate **true** and **false** for the operands in each condition. The result of the condition is shown as **true** or **false**. Note that when you concatenate a **boolean** value with a **String**, Java automatically adds the string “false” or “true,” based on the **boolean** value.

Line 13 in method **main** creates a **JTextArea**. The numbers in the parentheses indicate that the **JTextArea** has 17 rows and 20 columns. Line 17 declares **JScrollPane** reference **scroller** and initializes it with a new **JScrollPane** object. Class

**JScrollPane** (from package `javax.swing`) provides a GUI component with scrolling functionality. A **JScrollPane** object is initialized with the GUI component for which it will provide scrolling functionality (**outputArea** in this example). This initialization attaches the GUI component to the **JScrollPane**. When you execute this application, notice the *scrollbar* on the right side of the **JTextArea**. You can click the *arrows* at the top or bottom of the scrollbar to scroll up or down, respectively, through the text in the **JTextArea** one line at a time. You can also drag the *scroll box* (also called the *thumb*) up or down to scroll through the text rapidly.

Lines 22–59 build the **output** string that is displayed in the **outputArea**. Line 61 uses method **setText** to replace the text in **outputArea** with the **output** string. Lines 63–64 display a message dialog. The second argument, **scroller**, indicates that the **scroller** (and the **outputArea** attached to it) should be displayed as the message in the message dialog.

The chart in Fig. 5.20 shows the precedence and associativity of the Java operators introduced up to this point. The operators are shown from top to bottom in decreasing order of precedence.

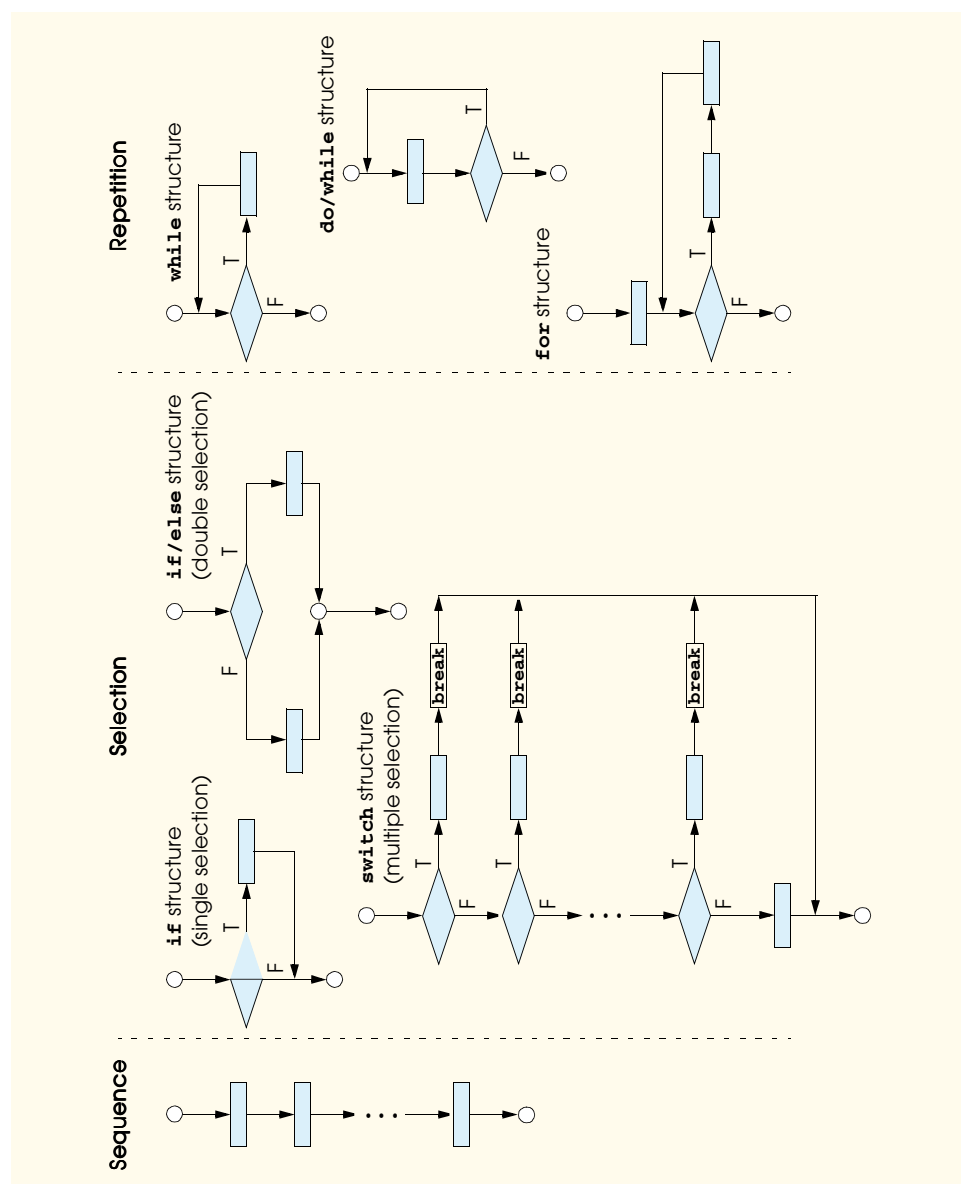
## 5.10 Structured Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture is, and our collective wisdom is considerably sparser. We have learned that structured programming produces programs that are easier than unstructured programs to understand and hence are easier to test, debug, modify and even prove correct in a mathematical sense.

| Operators          | Associativity | Type                         |
|--------------------|---------------|------------------------------|
| ()                 | left to right | parentheses                  |
| ++ --              | right to left | unary postfix                |
| ++ -- + - ! (type) | right to left | unary                        |
| * / %              | left to right | multiplicative               |
| + -                | left to right | additive                     |
| < <= > >=          | left to right | relational                   |
| == !=              | left to right | equality                     |
| &                  | left to right | boolean logical AND          |
| ^                  | left to right | boolean logical exclusive OR |
|                    | left to right | boolean logical inclusive OR |
| &&                 | left to right | logical AND                  |
|                    | left to right | logical OR                   |
| ?:                 | right to left | conditional                  |
| = += -= *= /= %=   | right to left | assignment                   |

**Fig. 5.20** Precedence and associativity of the operators discussed so far.

Figure 5.21 summarizes Java's control structures. Small circles are used in the figure to indicate the single entry and single exit points of each structure. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control structures, and to build structured programs by properly combining control structures in two simple ways.



**Fig. 5.21** Java's single-entry/single-exit control structures.

For simplicity, only single-entry/single-exit control structures are used; there is only one way to enter and only one way to exit each control structure. Connecting control structures in sequence to form structured programs is simple: The exit point of one control structure is connected to the entry point of the next control structure (i.e., the control structures are simply placed one after another in a program); we have called this method “control-structure stacking.” The rules for forming structured programs also allow for control-structure nesting.

Figure 5.22 shows the rules for forming properly structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action, including input/output.

Applying the rules of Fig. 5.22 always results in a structured flowchart with a neat, building-block-like appearance (Fig. 5.23). For example, repeatedly applying Rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence (Fig. 5.24). Rule 2 generates a stack of control structures; so let us call Rule 2 the *stacking rule*. [Note: The symbols at the top and bottom of Fig. 5.23 represent the beginning and end of a program, respectively.]

Rule 3 is called the *nesting rule*. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control structures. For example, in Fig. 5.25, the rectangle in the simplest flowchart is first replaced with a double-selection (**if/else**) structure. Then, Rule 3 is applied again to both of the rectangles in the double-selection structure, replacing each of the rectangles with double-selection structures. The dashed boxes around each of the double-selection structures represent the rectangle that was replaced with a double-selection structure.

#### Rules for Forming Structured Programs

- 1) Begin with the “simplest flowchart” (Fig. 5.23).
- 2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
- 3) Any rectangle (action) can be replaced by any control structure (sequence, **if**, **if/else**, **switch**, **while**, **do/while** or **for**).
- 4) Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 5.22 Rules for forming structured programs.

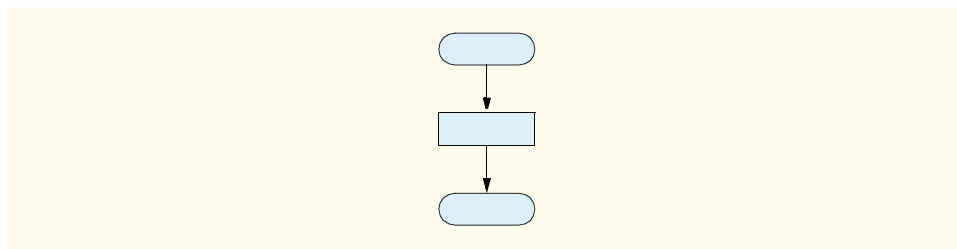
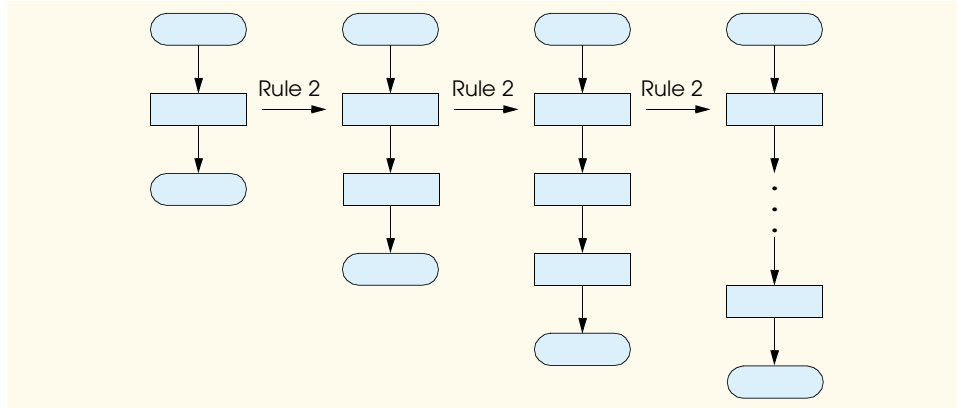
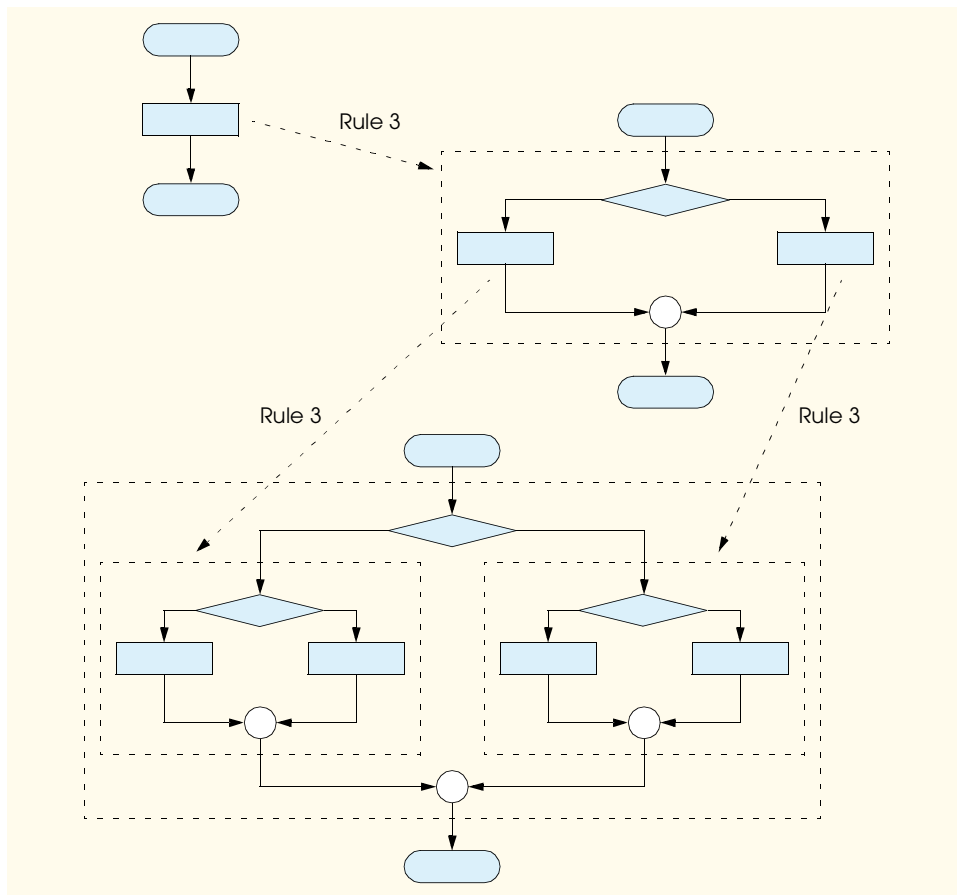


Fig. 5.23 The simplest flowchart.



**Fig. 5.24** Repeatedly applying Rule 2 of Fig. 5.22 to the simplest flowchart.



**Fig. 5.25** Applying Rule 3 of Fig. 5.22 to the simplest flowchart.

Rule 4 generates larger, more involved and more deeply nested structures. The flowcharts that emerge from applying the rules in Fig. 5.22 constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

The beauty of the structured approach is that we use only seven simple single-entry/single-exit pieces and we assemble them in only two simple ways. Figure 5.26 shows the kinds of stacked building blocks that emerge from applying Rule 2 and the kinds of nested building blocks that emerge from applying Rule 3. The figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the **goto** statement).

If the rules in Fig. 5.22 are followed, you cannot create an unstructured flowchart (such as that in Fig. 5.27). If you are uncertain if a particular flowchart is structured, apply the rules of Fig. 5.22 in reverse to try to reduce the flowchart to the simplest flowchart. If the flowchart is reducible to the simplest flowchart, the original flowchart is structured; otherwise, it is not.

Structured programming promotes simplicity. Bohm and Jacopini have shown that only three forms of control are needed—sequence, selection and repetition.

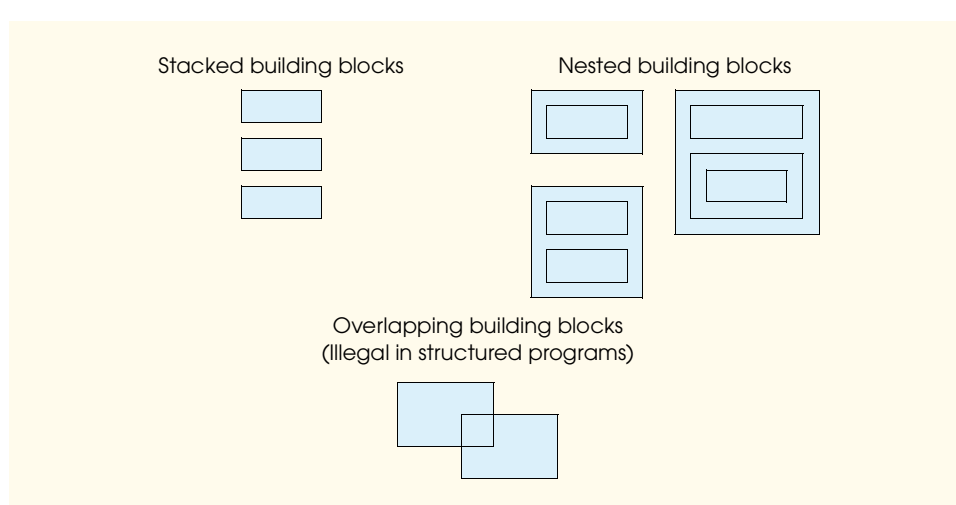


Fig. 5.26 Stacked, nested and overlapped building blocks.

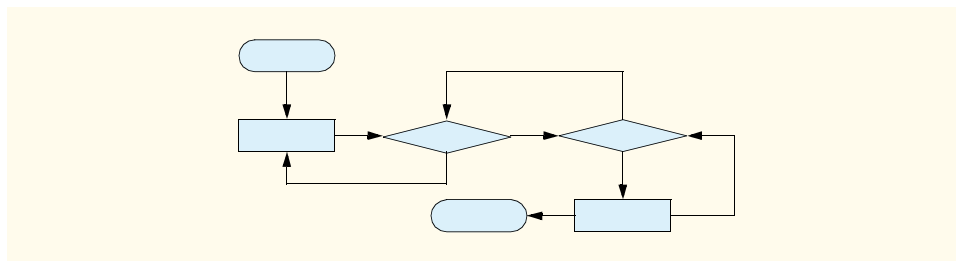


Fig. 5.27 An unstructured flowchart.

Sequence is trivial. Selection is implemented in one of three ways:

- an **if** structure (single selection),
- an **if/else** structure (double selection), or
- a **switch** structure (multiple selection).

In fact, it is straightforward to prove that the **if** structure is sufficient for any form of selection; everything that can be done with the **if/else** structure and the **switch** structure can be implemented by combining **if** structures (although perhaps not as elegantly).

Repetition is implemented in one of three ways:

- a **while** structure,
- a **do/while** structure, or
- a **for** structure.

It is straightforward to prove that the **while** structure is sufficient to provide any form of repetition. Everything that can be done with the **do/while** structure and the **for** structure can be done with the **while** structure (although perhaps not as elegantly).

Combining these results illustrates that any form of control ever needed in a Java program can be expressed in terms of

- a sequence,
- a **if** structure (selection), or
- a **while** structure (repetition).

And these control structures can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

In this chapter, we have discussed how to compose programs from control structures containing actions and decisions. In Chapter 6, we introduce another program structuring unit, called the *method*. We will learn to compose large programs by combining methods that, in turn, are composed of control structures. We will also discuss how methods promote software reusability. In Chapter 8, we discuss in more detail Java's other program-structuring unit, called the *class*. We will then create objects from classes and proceed with our treatment of object-oriented programming.

## 5.11 (Optional Case Study) Thinking About Objects: Identifying Objects' States and Activities

In “Thinking About Objects,” Section 4.14, we determined many of the class attributes needed to implement the elevator simulator and added them to the class diagram of Fig. 4.18. In this section, we show how these attributes represent an object's *state*, or condition. We identify the set of possible states that our objects may occupy and discuss how these objects change state in response to *messages*. We also discuss the workflow, or the *activities*, that an object performs in our elevator simulation.

### Statechart Diagrams

Objects in a system have *states*. A state describes the condition of an object at a given time. *Statechart diagrams* (also called *state diagrams*) give us a way to express how, and under what conditions, the objects in a system change state. Unlike the class and object

diagrams presented in earlier case-study sections, statechart diagrams model the behavior of the system.

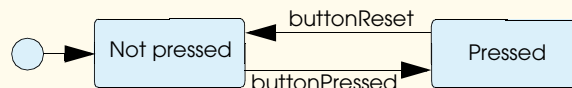
Figure 5.28 is a simple statechart diagram that models the states of an object of either class **FloorButton** or class **ElevatorButton**. The UML represents each state in a statechart diagram as a *rounded rectangle* with the name of the state placed inside the rectangle. A *solid circle* with an attached arrowhead designates the initial state (in this case, the “Not pressed” state). Notice that this statechart diagram models the **boolean** attribute **pressed** in the class diagram of Fig. 4.18. The attribute is initialized to **false**, or the “Not pressed” state according to the statechart diagram.

The arrows indicate *transitions* between states. An object can transition from one state to another in response to a *message*. For example, the **FloorButton** and **ElevatorButton** objects change from the “Not pressed” state to the “Pressed” state in response to a **buttonPressed** message, and the **pressed** attribute changes to a value of **true**. The name of the message that causes a transition is written near the line that corresponds to that transition. (We explain messages in Section 7.10 and Section 10.22.)

Objects from other classes, such as **Light**, **Elevator** and **Person**, have similar statechart diagrams. Class **Light** has an “on” state or an “off” state—transitions between these states occur as a result of “turn on” and “turn off” events, respectively. Class **Elevator** and class **Person** each have a “moving” state and a “waiting” state—transitions between these states occur as a result of “start moving” and “stop moving” events, respectively.

### Activity Diagrams

The *activity diagram* is similar to the statechart diagram in that they both model aspects of system behavior. Unlike a statechart diagram, an activity diagram models an object’s workflow during program execution. An activity diagram is a flowchart that models the *actions* the object will perform and in what order. The activity diagram in Fig. 5.29 models the activities of a person. The diagram begins with the person moving toward the floor button. If the door is open, the person waits for the current elevator passenger (if one exists) to exit then enters the elevator.<sup>1</sup> If the door is closed, the person presses the floor button and waits for the elevator to open the door. When the door opens, the person waits for the elevator passenger to exit (if one exists) then enters the elevator. The person presses the elevator button, which causes the elevator to move to the other floor, unless the elevator already services that floor; the person then waits for the doors to re-open and exits the elevator after the doors open.



**Fig. 5.28** Statechart diagram for **FloorButton** and **ElevatorButton** objects.

1. We use *multithreading* and *synchronized methods* in Section 15.12 to guarantee that the passenger riding the elevator will exit before the person waiting for the elevator will enter.

The UML represents activities as ovals in activity diagrams. The name of the activity is placed inside the oval. An arrow connects two activities, indicating the order in which the activities are performed. As with statechart diagrams, the *solid circle* indicates the starting activity. In this case, the person moving toward the floor button is the starting activity. The activity flow arrives at a *branch* (indicated by the *small diamond symbol*) after the person moves to the floor button. This point determines the next activity based on the associated *guard condition* (in square brackets above the transition), which states that the transition occurs if this condition is met. For example, in Fig. 5.29, if the floor door is closed, the person presses the floor button, waits for the door to open, waits for the passenger (if there is one) to exit the elevator, then enters the elevator. However, if the floor door is open, the person waits for the passenger (if there is one) to exit the elevator, then enters the elevator. Regardless of whether the floor door was open or closed at the last diamond symbol, the person now presses the elevator button (which causes the doors to close and the elevator to move to the other floor), the person waits for the elevator door to open—when this door opens, the person exits the elevator. Activity diagrams are similar to the flowcharts for control structures presented in Chapters 4 and 5—both diagram types employ diamond symbols to alter the flow of control between activities.

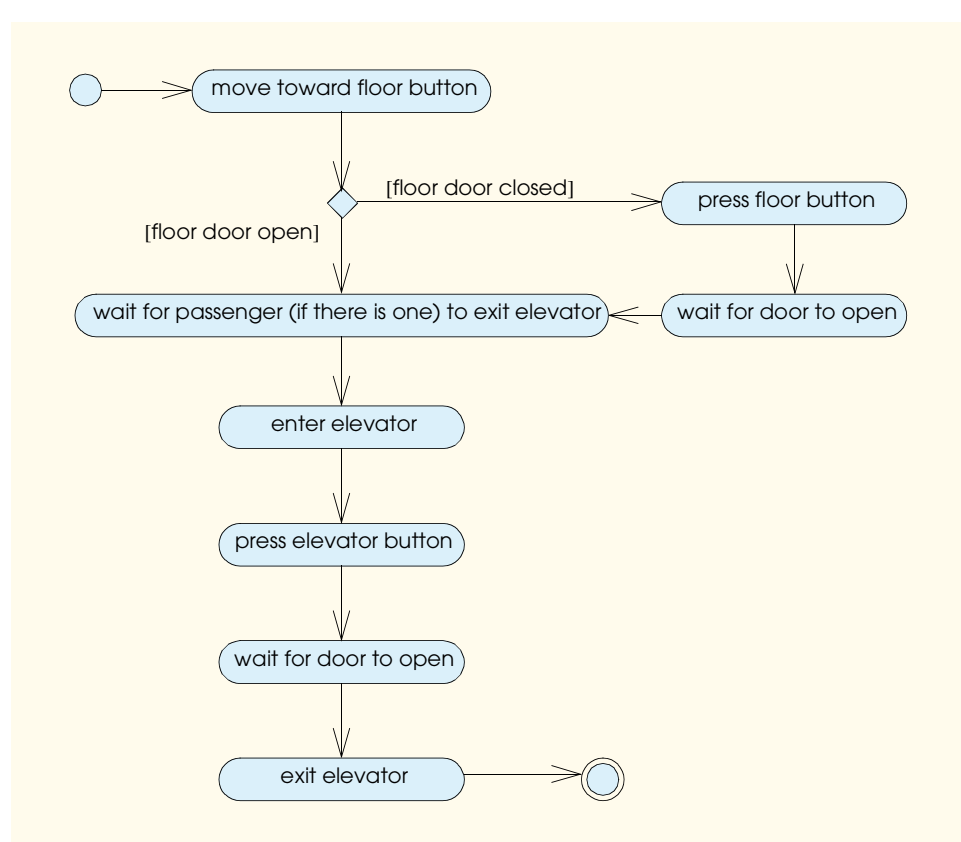


Fig. 5.29 Activity diagram for a **Person** object.

Figure 5.30 shows an activity diagram for the elevator. The diagram begins when a button is pressed. If the button is an elevator button, the elevator sets **summoned** to false (we explain this **boolean** variable in a moment), closes the elevator door, moves to the other floor, resets the elevator button, rings the bell and opens the elevator door. If the button is a floor button, the next branch determines the next transition, based on whether the elevator is moving. If the elevator is idle, the next branch determines which floor button generated the request. If the request originated from the current floor on which the elevator is located, the elevator resets its button, rings its bell and opens its door. If the request originated from the opposite floor, the elevator closes the door and moves to the opposite (destination) floor, where the elevator resets its button, rings its bell and opens its door. Now consider the activity if the elevator is moving. A separate branch determines which floor button generated the request. If the request originated from the destination floor, the elevator continues traveling to that floor. If the request originated from the floor from which the elevator departed, the elevator continues traveling to the destination floor, but must remember to return to the requesting floor. The **summoned** attribute, originally displayed in Fig. 4.18, is set to **true** so that the elevator knows to return to the other floor after the elevator opens its door.

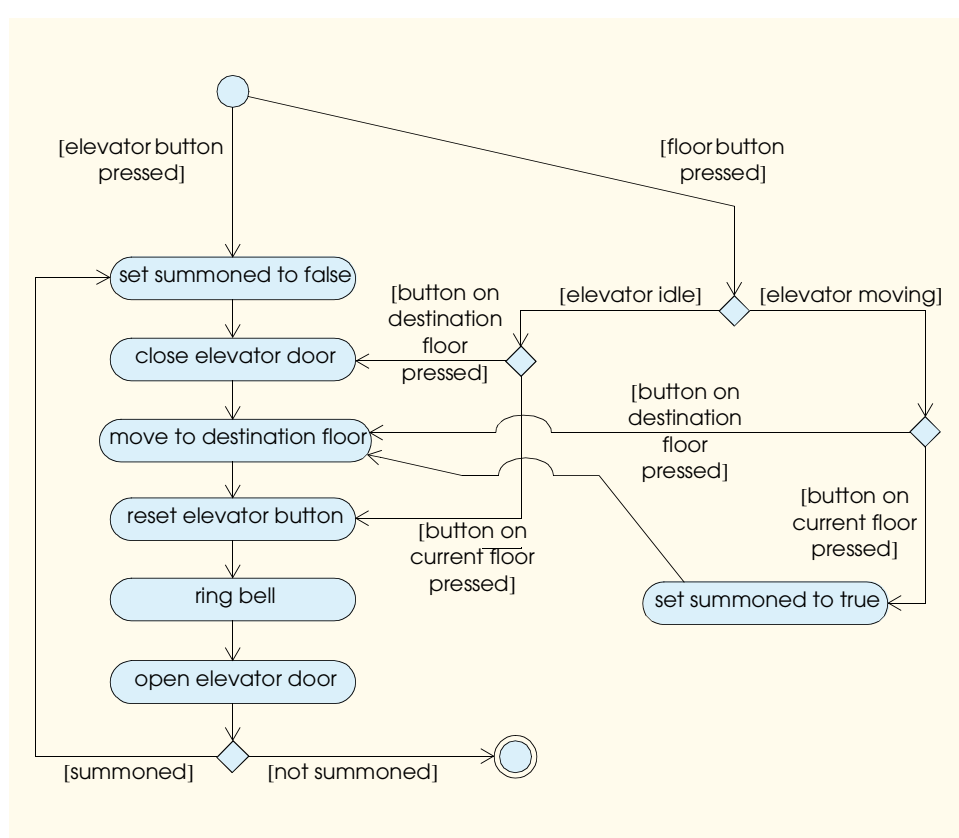


Fig. 5.30 Activity diagram for the **Elevator** object.

We have taken the first steps to modeling the behavior of the system and have shown how the attributes of an object determine that object's activity. In "Thinking About Objects," Section 6.17, we investigate the behaviors for all classes to give a more accurate interpretation of the system behavior by "filling in" the final compartment for the classes in our class diagram.

## SUMMARY

- The **for** repetition structure handles all of the details of counter-controlled repetition. The general format of the **for** structure is

```
for (expression1; expression2; expression3)
    statement
```

where *expression1* initializes the loop's control variable, *expression2* is the loop-continuation condition and *expression3* modifies the control variable, so that the loop-continuation condition eventually becomes **false**.

- A **JTextArea** is a GUI component that is capable of displaying many lines of text.
- Method **setText** replaces the text in a **JTextArea**. Method **append** adds text to the end of the text in a **JTextArea**.
- **NumberFormat** static method **getCurrencyInstance** returns a **NumberFormat** object that can format numeric values as currency. The argument **Locale.US** indicates that the currency values should be displayed starting with a dollar sign (\$), use a decimal point to separate dollars and cents and use a comma to delineate thousands.
- Class **Locale** provides constants that can be used to customize programs to represent currency values for other countries, so that currency formats are displayed properly for each locale.
- Class **NumberFormat** is located in package **java.text**.
- Class **Locale** is located in package **java.util**.
- An interesting feature of class **JOptionPane** is that the message it displays with **showMessageDialog** can be a **String** or a GUI component, such as a **JTextArea**.
- The **switch** structure handles a series of decisions in which a particular variable or expression is tested for values it may assume, and different actions are taken. In most programs, it is necessary to include a **break** statement after the statements for each **case**. Several **cases** can execute the same statements by listing the **case** labels together before the statements. The **switch** structure can only test for constant integral expressions.
- The **do/while** repetition structure tests the loop-continuation condition at the end of the loop, so the body of the loop will be executed at least once. The format for the **do/while** structure is

```
do {
    statement
} while (condition);
```

- The **break** statement, when executed in one of the repetition structures (**for**, **while** and **do/while**), causes immediate exit from the structure.
- The **continue** statement, when executed in one of the repetition structures (**for**, **while** and **do/while**), skips any remaining statements in the body of the structure and proceeds with the test for the next iteration of the loop.
- To break out of a nested set of structures, use the labeled **break** statement. This statement, when executed in a **while**, **for**, **do/while** or **switch** structure, causes immediate exit from that

structure and any number of enclosing repetition structures; program execution resumes with the first statement after the enclosing labeled block.

- The labeled **continue** statement, when executed in a repetition structure (**while**, **for** or **do/while**), skips the remaining statements in that structure's body and any number of enclosing repetition structures and proceeds with the next iteration of the enclosing labeled repetition structure.
- Logical operators may be used to form complex conditions by combining conditions. The logical operators are **&&**, **&**, **||**, **|**, **^** and **!**, meaning logical AND, boolean logical AND, logical OR, boolean logical inclusive OR, boolean logical exclusive OR and logical NOT (negation), respectively.
- Class **JScrollPane** provides a GUI component with scrolling functionality.

## TERMINOLOGY

|                                                |                                             |
|------------------------------------------------|---------------------------------------------|
| <b>!</b> operator                              | labeled repetition structure                |
| <b>&amp;&amp;</b> operator                     | <b>Locale</b> class                         |
| <b>  </b> operator                             | <b>Locale.US</b>                            |
| <b>append</b> method of class <b>JTextArea</b> | logical AND ( <b>&amp;&amp;</b> )           |
| boolean logical AND ( <b>&amp;</b> )           | logical negation ( <b>!</b> )               |
| boolean logical exclusive OR ( <b>^</b> )      | logical operators                           |
| boolean logical inclusive OR ( <b> </b> )      | logical OR ( <b>  </b> )                    |
| <b>break</b>                                   | long                                        |
| <b>case</b> label                              | loop-continuation condition                 |
| <b>continue</b>                                | multiple selection                          |
| counter-controlled repetition                  | nested control structures                   |
| <b>default</b> case in <b>switch</b>           | <b>NumberFormat</b> class                   |
| definite repetition                            | off-by-one error                            |
| <b>do/while</b> repetition structure           | repetition structures                       |
| <b>for</b> repetition structure                | scroll box                                  |
| infinite loop                                  | scrollbar                                   |
| <b>java.text</b> package                       | short-circuit evaluation                    |
| <b>java.util</b> package                       | single-entry/single-exit control structures |
| <b>JScrollPane</b> class                       | stacked control structures                  |
| <b>JTextArea</b> class                         | <b>switch</b> selection structure           |
| labeled <b>break</b> statement                 | thumb of a scrollbar                        |
| labeled block                                  | <b>while</b> repetition structure           |
| labeled <b>continue</b> statement              |                                             |

## SELF-REVIEW EXERCISES

- 5.1** State whether each of the following is true or false. If false, explain why.
- The **default** case is required in the **switch** selection structure.
  - The **break** statement is required in the default case of a **switch** selection structure.
  - The expression **( x > y && a < b )** is true if either **x > y** is true or **a < b** is true.
  - An expression containing the **||** operator is true if either or both of its operands is true.
- 5.2** Write a Java statement or a set of Java statements to accomplish each of the following tasks:
- Sum the odd integers between 1 and 99, using a **for** structure. Assume that the integer variables **sum** and **count** have been declared.
  - Calculate the value of **2.5** raised to the power of **3**, using the **pow** method.
  - Print the integers from 1 to 20, using a **while** loop and the counter variable **x**. Assume that the variable **x** has been declared, but not initialized. Print only five integers per line. [Hint: Use the calculation **x % 5**. When the value of this expression is 0, print a newline character; otherwise, print a tab character. Assume that this code is an application; use

the `System.out.println()` method to output the newline character, and use the `System.out.print( '\t' )` method to output the tab character.]

d) Repeat Exercise 5.2 (c), using a **for** structure.

**5.3** Find the error in each of the following code segments, and explain how to correct it:

a) `x = 1;`

```
while ( x <= 10 );
    x++;
}
```

b) `for ( y = .1; y != 1.0; y += .1 )`  
`System.out.println( y );`

c) `switch ( n ) {`  
`case 1:`  
`System.out.println( "The number is 1" );`  
`case 2:`  
`System.out.println( "The number is 2" );`  
`break;`  
`default:`  
`System.out.println( "The number is not 1 or 2" );`  
`break;`  
`}`

d) The following code should print the values 1 to 10.

```
n = 1;

while ( n < 10 )
    System.out.println( n++ );
```

## ANSWERS TO SELF-REVIEW EXERCISES

**5.1** a) False. The **default** case is optional. If no default action is needed, then there is no need for a **default** case. b) False. The **break** statement is used to exit the **switch** structure. The **break** statement is not required for the last case in a **switch** structure. c) False. Both of the relational expressions must be true for the entire expression to be true when using the **&&** operator. d) True.

**5.2** The answers to Exercise 5.2 are as follows:

a) `sum = 0;`  
`for ( count = 1; count <= 99; count += 2 )`  
`sum += count;`  
b) `Math.pow( 2.5, 3 )`  
c) `x = 1;`

```
while ( x <= 20 ) {
    System.out.print( x );

    if ( x % 5 == 0 )
        System.out.println();
    else
        System.out.print( '\t' );

    ++x;
}
```

```
d) for ( x = 1; x <= 20; x++ ) {
    System.out.print( x );

    if ( x % 5 == 0 )
        System.out.println();
    else
        System.out.print( '\t' );
}

or

for ( x = 1; x <= 20; x++ )

    if ( x % 5 == 0 )
        System.out.println( x );
    else
        System.out.print( x + "\t" );
```

**5.3** The answers to Exercise 5.3 are as follows:

- Error: The semicolon after the **while** header causes an infinite loop, and there is a missing left brace.  
Correction: Replace the semicolon by a **{**, or remove both the **;** and the **}**.
- Error: Using a floating-point number to control a **for** repetition structure may not work, because floating-point numbers are represented only approximately by most computers.  
Correction: Use an integer, and perform the proper calculation in order to get the values you desire:  

```
for ( y = 1; y != 10; y++ )
    System.out.println( ( float ) y / 10 );
```
- Error: Missing **break** statement in the statements for the first **case**.  
Correction: Add a **break** statement at the end of the statements for the first **case**. Note that this omission is not necessarily an error if the programmer wants the statement of **case 2:** to execute every time the **case 1:** statement executes.
- Error: Improper relational operator used in the **while** repetition-continuation condition.  
Correction: Use **<=** rather than **<**, or change **10** to **11**.

## EXERCISES

**5.4** Find the error(s) in each of the following segments of code:

- ```
For ( x = 100, x >= 1, x++ )
    System.out.println( x );
```
- The following code should print whether integer **value** is odd or even:  

```
switch ( value % 2 ) {

    case 0:
        System.out.println( "Even integer" );

    case 1:
        System.out.println( "Odd integer" );
}
```
- The following code should output the odd integers from 19 to 1:  

```
for ( x = 19; x >= 1; x += 2 )
    System.out.println( x );
```

d) The following code should output the even integers from 2 to 100:

```
counter = 2;

do {
    System.out.println( counter );
    counter += 2;
} while ( counter < 100 );
```

5.5 What does the following program do?

```
1 public class Printing {
2
3     public static void main( String args[] )
4     {
5         for ( int i = 1; i <= 10; i++ ) {
6
7             for ( int j = 1; j <= 5; j++ )
8                 System.out.print( '@' );
9
10            System.out.println();
11        }
12    }
13
14 }
15
16 }
```

5.6 Write an application that finds the smallest of several integers. Assume that the first value read specifies the number of values to input from the user.

5.7 Write an application that calculates the product of the odd integers from 1 to 15, and then displays the results in a message dialog.

5.8 The *factorial* method is used frequently in probability problems. The factorial of a positive integer  $n$  (written  $n!$  and pronounced “ $n$  factorial”) is equal to the product of the positive integers from 1 to  $n$ . Write an application that evaluates the factorials of the integers from 1 to 5. Display the results in tabular format in a **JTextArea** that is displayed on a message dialog. What difficulty might prevent you from calculating the factorial of 20?

5.9 Modify the compound-interest program of Fig. 5.6 to repeat its steps for interest rates of 5, 6, 7, 8, 9 and 10%. Use a **for** loop to vary the interest rate. Add scrolling functionality to the **JTextArea**, so you can scroll through all the output.

5.10 Write an application that displays the following patterns separately one below the other. Use **for** loops to generate the patterns. All asterisks (\*) should be printed by a single statement of the form **System.out.print( '\*' );**. (This statement causes the asterisks to print side by side.) A statement of the form **System.out.println();** can be used to position to the next line. A statement of the form **System.out.print( ' ' );** can be used to display a space for the last two patterns. There should be no other output statements in the program. [*Hint:* The last two patterns require that each line begin with an appropriate number of blank spaces.]





**5.21** Modify the program you wrote in Exercise 5.20 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

**5.22** A criticism of the **break** statement and the **continue** statement is that each is unstructured. Actually, **break** statements and **continue** statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you would remove any **break** statement from a loop in a program and replace that statement with some structured equivalent. [Hint: The **break** statement leaves a loop from within the body of the loop. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates “early exit because of a ‘break’ condition.”] Use the technique you developed here to remove the **break** statement from the program in Fig. 5.11.

**5.23** What does the following program segment do?

```
for ( i = 1; i <= 5; i++ ) {  
    for ( j = 1; j <= 3; j++ ) {  
        for ( k = 1; k <= 4; k++ )  
            System.out.print( ' * ' );  
        System.out.println();  
    }  
    System.out.println();  
}
```

**5.24** Describe in general how you would remove any **continue** statement from a loop in a program and replace that statement with some structured equivalent. Use the technique you developed here to remove the **continue** statement from the program in Fig. 5.12.

**5.25** (“The Twelve Days of Christmas” Song) Write an application that uses repetition and **switch** structures to print the song “The Twelve Days of Christmas.” One **switch** structure should be used to print the day (i.e., “First,” “Second,” etc.). A separate **switch** structure should be used to print the remainder of each verse. Visit the Web site [www.12days.com/library/carols/12daysofXmas.htm](http://www.12days.com/library/carols/12daysofXmas.htm) for the complete lyrics to the song.