# Control Statements: Part 2; Logical Operators

# 4

*The wheel is come full circle.*
—William Shakespeare

*All the evolution we know of proceeds from the vague to the definite.*
—Charles Sanders Peirce

## Objectives

In this chapter you'll:

- Learn the essentials of counter-controlled repetition.
- Use the `for` and `do...while` repetition statements to execute statements in a program repeatedly.
- Understand multiple selection using the `switch` selection statement.
- Use the `break` and `continue` program control statements to alter the flow of control.
- Use the logical operators to form complex conditional expressions in control statements.

Outline



## 4.1 Introduction

This chapter continues our presentation of structured programming theory and principles by introducing all but one of Java's remaining control statements. We demonstrate Java's `for`, `do...while` and `switch` statements. Through a series of short examples using `while` and `for`, we explore the essentials of counter-controlled repetition. We use a `switch` statement to count the number of A, B, C, D and F grade equivalents in a set of numeric grades entered by the user. We introduce the `break` and `continue` program-control statements. We discuss Java's logical operators, which enable you to use more complex conditional expressions in control statements. Finally, we summarize Java's control statements and the proven problem-solving techniques presented in this chapter and Chapter 3.

## 4.2 Essentials of Counter-Controlled Repetition

This section uses the `while` repetition statement introduced in Chapter 3 to formalize the elements required to perform counter-controlled repetition, which requires

1. a **control variable** (or loop counter)
2. the **initial value** of the control variable
3. the **increment** by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
4. the **loop-continuation condition** that determines if looping should continue.

To see these elements of counter-controlled repetition, consider the application of Fig. 4.1, which uses a loop to display the numbers from 1 through 10.

```
1  // Fig. 4.1: WhileCounter.java
2  // Counter-controlled repetition with the while repetition statement.
3
4  public class WhileCounter
5  {
6     public static void main(String[] args)
7     {
```

**Fig. 4.1** | Counter-controlled repetition with the `while` repetition statement. (Part 1 of 2.)

```
 8         int counter = 1; // declare and initialize control variable
 9
10         while (counter <= 10) // loop-continuation condition
11         {
12            System.out.printf("%d  ", counter);
13            ++counter; // increment control variable
14         }
15
16         System.out.println();
17      }
18   } // end class WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

**Fig. 4.1** | Counter-controlled repetition with the `while` repetition statement. (Part 2 of 2.)

In Fig. 4.1, the elements of counter-controlled repetition are defined in lines 8, 10 and 13. Line 8 *declares* the control variable (`counter`) as an `int`, *reserves space* for it in memory and sets its *initial value* to `1`. Variable `counter` also could have been declared and initialized with the following local-variable declaration and assignment statements:

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

Line 12 displays control variable `counter`'s value during each iteration of the loop. Line 13 *increments* the control variable by 1 for each iteration of the loop. The loop-continuation condition in the `while` (line 10) tests whether the value of the control variable is less than or equal to `10` (the final value for which the condition is `true`). The program performs the body of this `while` even when the control variable is `10`. The loop terminates when the control variable exceeds `10` (i.e., `counter` becomes `11`).

### Common Programming Error 4.1

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

### Error-Prevention Tip 4.1

*Use integers to control counting loops.*

The program in Fig. 4.1 can be made more concise by initializing `counter` to `0` in line 8 and *preincrementing* `counter` in the `while` condition as follows:

```
while (++counter <= 10) // loop-continuation condition
   System.out.printf("%d  ", counter);
```

This code saves a statement, because the `while` condition performs the increment before testing the condition. (Recall from Section 3.12 that the precedence of `++` is higher than that of `<=`.) Coding in such a condensed fashion takes practice, might make code more difficult to read, debug, modify and maintain, and typically should be avoided.

### Software Engineering Observation 4.1

*"Keep it simple" is good advice for most of the code you'll write.*

## 4.3 for Repetition Statement

Section 4.2 presented the essentials of counter-controlled repetition. The `while` statement can be used to implement any counter-controlled loop. Java also provides the **for repetition statement**, which specifies the counter-controlled-repetition details in a single line of code. Figure 4.2 reimplements the application of Fig. 4.1 using `for`.

```
1  // Fig. 4.2: ForCounter.java
2  // Counter-controlled repetition with the for repetition statement.
3
4  public class ForCounter
5  {
6     public static void main(String[] args)
7     {
8        // for statement header includes initialization,
9        // loop-continuation condition and increment
10       for (int counter = 1; counter <= 10; counter++)
11          System.out.printf("%d  ", counter);
12
13       System.out.println();
14    }
15 } // end class ForCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

**Fig. 4.2** | Counter-controlled repetition with the `for` repetition statement.

When the `for` statement (lines 10–11) begins executing, the control variable `counter` is *declared* and *initialized* to `1`. (Recall from Section 4.2 that the first two elements of counter-controlled repetition are the *control variable* and its *initial value*.) Next, the program checks the *loop-continuation condition*, `counter <= 10`, which is between the two required semicolons. Because the initial value of `counter` is `1`, the condition initially is true. Therefore, the body statement (line 11) displays control variable `counter`'s value, namely `1`. After executing the loop's body, the program increments `counter` in the expression `counter++`, which appears to the right of the second semicolon. Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop. At this point, the control variable's value is `2`, so the condition is still true (the *final value* is not exceeded)—thus, the program performs the body statement again (i.e., the next iteration of the loop). This process continues until the numbers 1 through 10 have been displayed and the `counter`'s value becomes `11`, causing the loop-continuation test to fail and repetition to terminate (after 10 repetitions of the loop body). Then the program performs the first statement after the `for`—in this case, line 13.

Figure 4.2 uses (in line 10) the loop-continuation condition `counter <= 10`. If you incorrectly specified `counter < 10` as the condition, the loop would iterate only nine times. This is a common *logic error* called an **off-by-one error**.

### Common Programming Error 4.2

*Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of a repetition statement can cause an off-by-one error.*

**Error-Prevention Tip 4.2**

*Using the final value and operator <= in a loop's condition helps avoid off-by-one errors. For a loop that outputs 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`.

**Error-Prevention Tip 4.3**

*As Chapter 3 mentioned, integers can overflow, causing logic errors. A loop's control variable also could overflow. Write your loop conditions carefully to prevent this.*

### *A Closer Look at the* `for` *Statement's Header*

Figure 4.3 takes a closer look at the `for` statement in Fig. 4.2. The first line—including the keyword `for` and everything in parentheses after `for` (line 10 in Fig. 4.2)—is sometimes called the **`for` statement header**. The `for` header "does it all"—it specifies each item needed for counter-controlled repetition with a control variable. If there's more than one statement in the body of the `for`, braces are required to define the body of the loop.

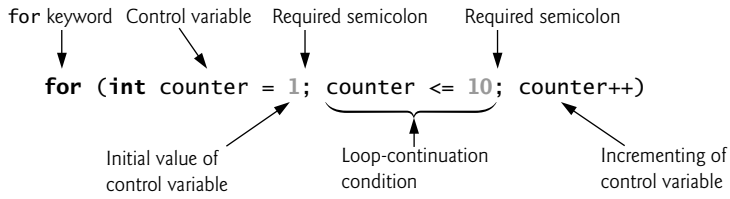


**Fig. 4.3** | `for` statement header components.

### *General Format of a* `for` *Statement*

The general format of the `for` statement is

```
for (initialization; loopContinuationCondition; increment)
   statement
```

where the *initialization* expression names the loop's control variable and *optionally* provides its initial value, *loopContinuationCondition* determines whether the loop should continue executing and *increment* modifies the control variable's value, so that the loop-continuation condition eventually becomes false. The two semicolons in the `for` header are required. If the loop-continuation condition is initially `false`, the program does *not* execute the `for` statement's body. Instead, execution proceeds with the statement following the `for`.

### *Representing a* `for` *Statement with an Equivalent* `while` *Statement*

The `for` statement often can be represented with an equivalent `while` statement as follows:

```
initialization;
while (loopContinuationCondition)
{
   statement
   increment;
}
```

In Section 4.7, we show a case in which a `for` statement cannot be represented with an equivalent `while` statement. Typically, `for` statements are used for counter-controlled repetition and `while` statements for sentinel-controlled repetition. However, `while` and `for` can each be used for either repetition type.

### *Scope of a `for` Statement's Control Variable*

If the *initialization* expression in the `for` header declares the control variable (i.e., the control variable's type is specified before the variable name, as in Fig. 4.2), the control variable can be used *only* in that `for` statement—it will not exist outside it. This restricted use is known as the variable's **scope**. The scope of a variable defines where it can be used in a program. For example, a *local variable* can be used *only* in the method that declares it and *only* from the point of declaration through the end of the method. Scope is discussed in detail in Chapter 5, Methods.

**Common Programming Error 4.3**

*When a `for` statement's control variable is declared in the initialization section of the `for`'s header, using the control variable after the `for`'s body is a compilation error.*

### *Expressions in a `for` Statement's Header Are Optional*

All three expressions in a `for` header are optional. If the *loopContinuationCondition* is omitted, Java assumes that the loop-continuation condition is *always true*, thus creating an *infinite loop*. You might omit the *initialization* expression if the program initializes the control variable *before* the loop. You might omit the *increment* expression if the program calculates the increment with statements in the loop's body or if no increment is needed. The increment expression in a `for` acts as if it were a standalone statement at the end of the `for`'s body. Therefore, the expressions

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

are equivalent increment expressions in a `for` statement. Many programmers prefer `counter++` because it's concise and because a `for` loop evaluates its increment expression *after* its body executes, so the postfix increment form seems more natural. In this case, the variable being incremented does not appear in a larger expression, so preincrementing and postincrementing actually have the *same* effect.

**Common Programming Error 4.4**

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

**Error-Prevention Tip 4.4**

*Infinite loops occur when the loop-continuation condition in a repetition statement never becomes `false`. To prevent this situation in a counter-controlled loop, ensure that the control variable is modified during each iteration of the loop so that the loop-continuation condition will eventually become `false`. In a sentinel-controlled loop, ensure that the sentinel value is able to be input.*

### *Placing Arithmetic Expressions in a* `for` *Statement's Header*

The initialization, loop-continuation condition and increment portions of a `for` statement 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` statement may also be *negative*, in which case it's a **decrement**, and the loop counts *downward*.

### *Using a* `for` *Statement's Control Variable in the Statement's Body*

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is *not* required. The control variable is commonly used to control repetition *without* being mentioned in the body of the `for`.

**Error-Prevention Tip 4.5**

*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.*

### *UML Activity Diagram for the* `for` *Statement*

The `for` statement's UML activity diagram is similar to that of the `while` statement (Fig. 3.4). Figure 4.4 shows the activity diagram of the `for` statement in Fig. 4.2. The diagram makes it clear that initialization occurs *once before* the loop-continuation test is evaluated the first time, and that incrementing occurs *each* time through the loop *after* the body statement executes.

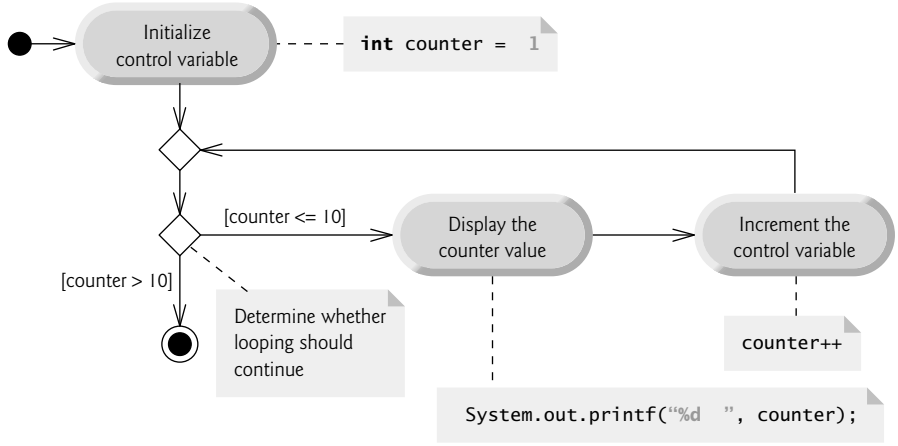


**Fig. 4.4** | UML activity diagram for the `for` statement in Fig. 4.2.

## 4.4 Examples Using the for Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write *only* the appropriate `for` header. Note the change in the relational operator for the 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 *decrements* of 1.

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

c) Vary the control variable from 7 to 77 in increments of 7.

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

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

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

e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

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

f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

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

### Common Programming Error 4.5

*Using an incorrect relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` instead of `i >= 1` in a loop counting down to 1) is usually a logic error.*

### Common Programming Error 4.6

*Do not use equality operators (`!=` or `==`) in a loop-continuation condition if the loop's control variable increments or decrements by more than 1. For example, consider the `for` statement header `for (int counter = 1; counter != 10; counter += 2)`. The loop-continuation test `counter != 10` never becomes false (resulting in an* infinite loop*) because `counter` increments by 2 after each iteration.*

### *Application: Summing the Even Integers from 2 to 20*

We now consider two sample applications that demonstrate simple uses of `for`. The application in Fig. 4.5 uses a `for` statement to sum the even integers from 2 to 20 and store the result in an `int` variable called `total`.

```
1  // Fig. 4.5: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum
5  {
```

**Fig. 4.5** | Summing integers with the `for` statement. (Part 1 of 2.)

```
 6     public static void main(String[] args)
 7     {
 8        int total = 0;
 9
10        // total even integers from 2 through 20
11        for (int number = 2; number <= 20; number += 2)
12           total += number;
13
14        System.out.printf("Sum is %d%n", total);
15     }
16  } // end class Sum
```

```
Sum is 110
```

**Fig. 4.5** | Summing integers with the `for` statement. (Part 2 of 2.)

The *initialization* and *increment* expressions can be comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions. For example, *although this is discouraged*, you could merge the body of the `for` statement in lines 11–12 of Fig. 4.5 into the increment portion of the `for` header by using a comma as follows:

```
for (int number = 2; number <= 20; total += number, number += 2)
   ; // empty statement
```

**Good Programming Practice 4.1**

*For readability limit the size of control-statement headers to a single line if possible.*

### *Application: Compound-Interest Calculations*

Let's use the `for` statement to compute compound interest. Consider the following problem:

> *A person invests $1,000 in a savings account yielding 5% interest. Assuming that all the 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 (e.g., use 0.05 for 5%)
> $n$ is the number of years
> $a$ is the amount on deposit at the end of the $n$th year.

The solution to this problem (Fig. 4.6) involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. Lines 8–10 in method `main` declare `double` variables `amount`, `principal` and `rate`, and initialize `principal` to `1000.0` and `rate` to `0.05`. Java treats floating-point constants like `1000.0` and `0.05` as type `double`. Similarly, Java treats whole-number constants like `7` and `-22` as type `int`.

```
1  // Fig. 4.6: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest
5  {
6     public static void main(String[] args)
7     {
8        double amount; // amount on deposit at end of each year
9        double principal = 1000.0; // initial amount before interest
10       double rate = 0.05; // interest rate
11
12       // display headers
13       System.out.printf("%s%20s%n", "Year", "Amount on deposit");
14
15       // calculate amount on deposit for each of ten years
16       for (int year = 1; year <= 10; ++year)
17       {
18          // calculate new amount for specified year
19          amount = principal * Math.pow(1.0 + rate, year);
20
21          // display the year and the amount
22          System.out.printf("%4d%,20.2f%n", year, amount);
23       }
24    }
25 } // end class Interest
```

```
Year   Amount on deposit
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89
```

**Fig. 4.6** | Compound-interest calculations with `for`.

*Formatting Strings with Field Widths and Justification*

Line 13 outputs the headers for two columns of output. The first column displays the year and the second column the amount on deposit at the end of that year. We use the format specifier `%20s` to output the `String` `"Amount on Deposit"`. The integer `20` between the `%` and the conversion character `s` indicates that the value should be displayed with a **field width** of 20—that is, `printf` displays the value with at least 20 character positions. If the value to be output is less than 20 character positions wide (17 characters in this example), the value is **right justified** in the field by default. If the `year` value to be output were more than four character positions wide, the field width would be extended to the right to accommodate the entire value—this would push the `amount` field to the right, upsetting the neat columns of our tabular output. To output values **left justified**, simply precede the field width with the **minus sign** (-) **formatting flag** (e.g., `%-20s`).

### *Performing the Interest Calculations with* `static` *Method* `pow` *of Class* `Math`

The `for` statement (lines 16–23) executes its body 10 times, varying control variable `year` from 1 to 10 in increments of 1. This loop terminates when `year` becomes 11. (Variable `year` represents $n$ in the problem statement.)

Classes provide methods that perform common tasks on objects. In fact, most methods must be called on a specific object. For example, to output text in Fig. 4.6, line 13 calls method `printf` on the `System.out` object, and to input `int` values from the user at the keyboard you called method `nextInt` on a `Scanner` object. Some classes also provide methods that perform common tasks and do *not* require objects. These are called `static` methods. For example, Java does *not* include an exponentiation operator, so the designers of Java's `Math` class defined `static` method `pow` for raising a value to a power. You can call a `static` method by specifying the *class name* followed by a dot (`.`) and the method name, as in

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

In Chapter 5, you'll learn how to implement `static` methods in your own classes.

We use `static` method **`pow`** of class **`Math`** to perform the compound-interest calculation in Fig. 4.6. `Math.pow(`$x$`,` $y$`)` calculates the value of $x$ raised to the $y^{\text{th}}$ power. The method receives two `double` arguments and returns a `double` value. Line 19 performs the calculation $a = p(1 + r)^n$, where $a$ is `amount`, $p$ is `principal`, $r$ is `rate` and $n$ is `year`. Class `Math` is defined in package `java.lang`, so you do *not* need to import class `Math` to use it.

The body of the `for` statement 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 it in every iteration of the loop is wasteful.

**Performance Tip 4.1**

*In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop. Many of today's sophisticated optimizing compilers will place such calculations outside loops in the compiled code.*

### *Formatting Floating-Point Numbers*

After each calculation, line 22 outputs the year and the amount on deposit at the end of that year. The year is output in a field width of four characters (as specified by `%4d`). The amount is output as a floating-point number with the format specifier `%,20.2f`. The **comma (`,`) formatting flag** indicates that the floating-point value should be output with a **grouping separator**. The actual separator used is specific to the user's locale (i.e., country). For example, in the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,234.45. The number `20` in the format specification indicates that the value should be output right justified in a *field width* of 20 characters. The `.2` specifies the formatted number's *precision*—in this case, the number is *rounded* to the nearest hundredth and output with two digits to the right of the decimal point.

### *A Warning about Displaying Rounded Values*

We declared variables `amount`, `principal` and `rate` to be of type `double` in this example. We're dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using `double` (or `float`) 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 output could appear as

```
  14.23
+ 18.67
-------
  32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

**Error-Prevention Tip 4.6**

*Do not use variables of type* `double` *(or* `float`*) to perform precise monetary calculations. The imprecision of floating-point numbers can lead to errors. In the exercises, you'll learn how to use integers to perform precise monetary calculations—Java also provides class* `java.math.BigDecimal` *for this purpose, which we demonstrate in Fig. 8.16.*

## 4.5 do...while Repetition Statement

The **`do...while` repetition statement** is similar to the `while` statement. In the `while`, the program tests the loop-continuation condition at the *beginning* of the loop, *before* executing the loop's body; if the condition is *false*, the body *never* executes. The `do...while` statement tests the loop-continuation condition *after* executing the loop's body; therefore, *the body always executes at least once*. When a `do...while` statement terminates, execution continues with the next statement in sequence. Figure 4.7 uses a `do...while` to output the numbers 1–10.

```
1  // Fig. 4.7: DoWhileTest.java
2  // do...while repetition statement.
3
4  public class DoWhileTest
5  {
6     public static void main(String[] args)
7     {
8        int counter = 1;
9
10       do
11       {
12          System.out.printf("%d  ", counter);
13          ++counter;
14       } while (counter <= 10); // end do...while
15
16       System.out.println();
17    }
18 } // end class DoWhileTest
```

**Fig. 4.7** | `do...while` repetition statement. (Part 1 of 2.)

```
1  2  3  4  5  6  7  8  9  10
```

**Fig. 4.7** | do...while repetition statement. (Part 2 of 2.)

Line 8 declares and initializes control variable `counter`. Upon entering the `do…while` statement, line 12 outputs `counter`'s value and line 13 increments `counter`. Then the program evaluates the loop-continuation test at the *bottom* of the loop (line 14). If the condition is *true*, the loop continues at the first body statement (line 12). If the condition is *false*, the loop terminates and the program continues at the next statement after the loop.

### *UML Activity Diagram for the* `do…while` *Repetition Statement*

Figure 4.8 contains the UML activity diagram for the `do…while` statement. This diagram makes it clear that the loop-continuation condition is not evaluated until *after* the loop performs the action state *at least once*. Compare this activity diagram with that of the `while` statement (Fig. 3.4).

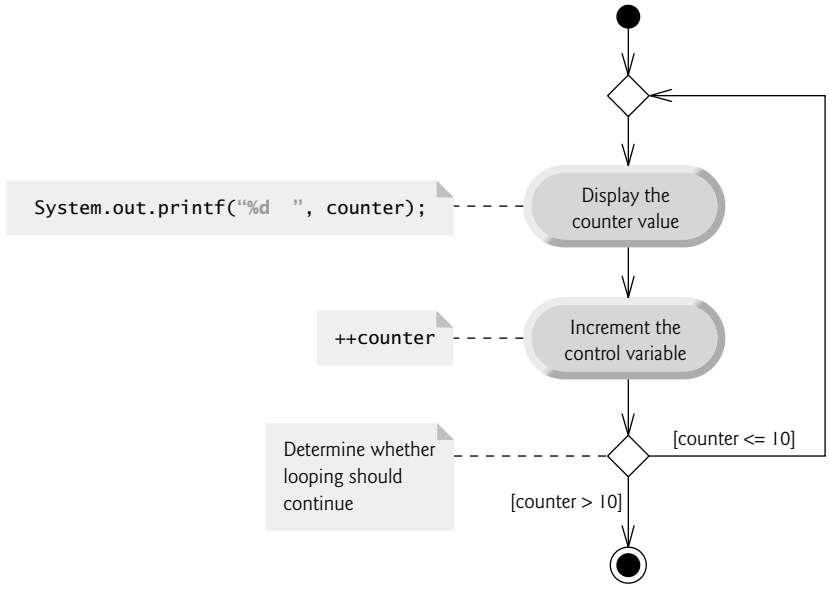


**Fig. 4.8** | do...while repetition statement UML activity diagram.

### *Braces in a* `do…while` *Repetition Statement*

It isn't necessary to use braces in the `do…while` repetition statement if there's only one statement in the body. However, many programmers include the braces, to avoid confusion between the `while` and `do…while` statements. For example,

```
while (condition)
```

is normally the first line of a `while` statement. A `do…while` statement with no braces around a single-statement body appears as:

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

which can be confusing. A reader may misinterpret the last line—`while(`*condition*`);`—as a `while` statement containing an empty statement (the semicolon by itself). Thus, the `do...while` statement with one body statement is usually written with braces as follows:

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

**Good Programming Practice 4.2**

*Always include braces in a `do...while` statement. This helps eliminate ambiguity between the `while` statement and a `do...while` statement containing only one statement.*

## 4.6 `switch` Multiple-Selection Statement

Chapter 3 discussed the `if` single-selection statement and the `if...else` double-selection statement. The **`switch` multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char`. As of Java SE 7, the expression may also be a `String`.

### *Using a `switch` Statement to Count A, B, C, D and F Grades*

Figure 4.9 calculates the class average of a set of numeric grades entered by the user, and uses a `switch` statement to determine whether each grade is the equivalent of an A, B, C, D or F and to increment the appropriate grade counter. The program also displays a summary of the number of students who received each grade.

Like earlier versions of the class-average program, the `main` method of class `LetterGrades` (Fig. 4.9) declares local variables `total` (line 9) and `gradeCounter` (line 10) to keep track of the sum of the grades entered by the user and the number of grades entered, respectively. Lines 11–15 declare counter variables for each grade category. Note that the variables in lines 9–15 are explicitly initialized to `0`.

Method `main` has two key parts. Lines 26–56 read an arbitrary number of integer grades from the user using sentinel-controlled repetition, update instance variables `total` and `gradeCounter`, and increment an appropriate letter-grade counter for each grade entered. Lines 59–80 output a report containing the total of all grades entered, the average of the grades and the number of students who received each letter grade. Let's examine these parts in more detail.

```
1  // Fig. 4.9: LetterGrades.java
2  // LetterGrades class uses the switch statement to count letter grades.
3  import java.util.Scanner;
4
5  public class LetterGrades
6  {
7     public static void main(String[] args)
8     {
```

**Fig. 4.9** | `LetterGrades` class uses the `switch` statement to count letter grades. (Part 1 of 3.)

```
 9         int total = 0; // sum of grades
10         int gradeCounter = 0; // number of grades entered
11         int aCount = 0; // count of A grades
12         int bCount = 0; // count of B grades
13         int cCount = 0; // count of C grades
14         int dCount = 0; // count of D grades
15         int fCount = 0; // count of F grades
16
17         Scanner input = new Scanner(System.in);
18
19         System.out.printf("%s%n%s%n   %s%n   %s%n",
20            "Enter the integer grades in the range 0-100.",
21            "Type the end-of-file indicator to terminate input:",
22            "On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter",
23            "On Windows type <Ctrl> z then press Enter");
24
25         // loop until user enters the end-of-file indicator
26         while (input.hasNext())
27         {
28            int grade = input.nextInt(); // read grade
29            total += grade; // add grade to total
30            ++gradeCounter; // increment number of grades
31
32            //  increment appropriate letter-grade counter
33            switch (grade / 10)
34            {
35               case 9:  // grade was between 90
36               case 10: // and 100, inclusive
37                  ++aCount;
38                  break; // exits switch
39
40               case 8: // grade was between 80 and 89
41                  ++bCount;
42                  break; // exits switch
43
44               case 7: // grade was between 70 and 79
45                  ++cCount;
46                  break; // exits switch
47
48               case 6: // grade was between 60 and 69
49                  ++dCount;
50                  break; // exits switch
51
52               default: // grade was less than 60
53                  ++fCount;
54                  break; // optional; exits switch anyway
55            } // end switch
56         } // end while
57
58         // display grade report
59         System.out.printf("%nGrade Report:%n");
60
```

**Fig. 4.9** | LetterGrades class uses the switch statement to count letter grades. (Part 2 of 3.)

```
61         // if user entered at least one grade...
62         if (gradeCounter != 0)
63         {
64            // calculate average of all grades entered
65            double average = (double) total / gradeCounter;
66
67            // output summary of results
68            System.out.printf("Total of the %d grades entered is %d%n",
69               gradeCounter, total);
70            System.out.printf("Class average is %.2f%n", average);
71            System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
72               "Number of students who received each grade:",
73               "A: ", aCount,   // display number of A grades
74               "B: ", bCount,   // display number of B grades
75               "C: ", cCount,   // display number of C grades
76               "D: ", dCount,   // display number of D grades
77               "F: ", fCount); // display number of F grades
78         } // end if
79         else // no grades were entered, so output appropriate message
80            System.out.println("No grades were entered");
81      } // end main
82   } // end class LetterGrades
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

**Fig. 4.9** | `LetterGrades` class uses the `switch` statement to count letter grades. (Part 3 of 3.)

### *Reading Grades from the User*

Lines 19–23 prompt the user to enter integer grades and to type the end-of-file indicator to terminate the input. The **end-of-file indicator** is a system-dependent keystroke combi-

nation which the user enters to indicate that there's *no more data to input*. In Chapter 15, Files, Streams and Object Serialization, you'll see how the end-of-file indicator is used when a program reads its input from a file.

On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence

*<Ctrl> d*

on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key. On Windows systems, end-of-file can be entered by typing

*<Ctrl> z*

[*Note:* On some systems, you must press *Enter* after typing the end-of-file key sequence. Also, Windows typically displays the characters `^Z` on the screen when the end-of-file indicator is typed, as shown in the output of Fig. 4.9.]

**Portability Tip 4.1**

*The keystroke combinations for entering end-of-file are system dependent.*

The `while` statement (lines 26–56) obtains the user input. The condition at line 26 calls `Scanner` method **`hasNext`** to determine whether there's more data to input. This method returns the `boolean` value `true` if there's more data; otherwise, it returns `false`. The returned value is then used as the value of the condition in the `while` statement. Method `hasNext` returns `false` once the user types the end-of-file indicator.

Line 28 inputs a grade value from the user. Line 29 adds `grade` to `total`. Line 30 increments `gradeCounter`. These variables are used to compute the average of the grades. Lines 33–55 use a `switch` statement to increment the appropriate letter-grade counter based on the numeric grade entered.

### *Processing the Grades*

The `switch` statement (lines 33–55) determines which counter to increment. We assume that the user enters a valid grade in the range 0–100. A grade in the range 90–100 represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D and 0–59 represents F. The `switch` statement consists of a block that contains a sequence of **`case` labels** and an optional **`default` case**. These are used in this example to determine which counter to increment based on the grade.

When the flow of control reaches the `switch`, the program evaluates the expression in the parentheses (`grade / 10`) following keyword `switch`. This is the `switch`'s **controlling expression**. The program compares this expression's value (which must evaluate to an integral value of type `byte`, `char`, `short` or `int`, or to a `String`) with each `case` label. The controlling expression in line 33 performs integer division, which *truncates the fractional part* of the result. Thus, when we divide a value from 0 to 100 by 10, the result is always a value from 0 to 10. We use several of these values in our `case` labels. For example, if the user enters the integer `85`, the controlling expression evaluates to 8. The `switch` compares `8` with each `case` label. If a match occurs (`case 8:` at line 40), the program executes that `case`'s statements. For the integer `8`, line 41 increments `bCount`, because a grade in the 80s is a B. The **`break` statement** (line 42) causes program control to proceed with the first statement after the `switch`—in this program, we reach the end of the `while` loop, so con-

trol returns to the loop-continuation condition in line 26 to determine whether the loop should continue executing.

The cases in our switch explicitly test for the values 10, 9, 8, 7 and 6. Note the cases at lines 35–36 that test for the values 9 and 10 (both of which represent the grade A). Listing cases consecutively in this manner with no statements between them enables the cases to perform the same set of statements—when the controlling expression evaluates to 9 or 10, the statements in lines 37–38 will execute. The switch statement does *not* provide a mechanism for testing *ranges* of values, so *every* value you need to test must be listed in a separate case label. Each case can have multiple statements. The switch statement differs from other control statements in that it does *not* require braces around multiple statements in a case.

### *case without a break Statement*

Without break statements, each time a match occurs in the switch, the statements for that case and subsequent cases execute until a break statement or the end of the switch is encountered. This is often referred to as "falling through" to the statements in subsequent cases. (This feature is perfect for writing a concise program that displays the iterative song "The Twelve Days of Christmas" in Exercise 4.29.)

**Common Programming Error 4.7**

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

### *The default Case*

If no match occurs between the controlling expression's value and a case label, the default case (lines 52–54) executes. We use the default case in this example to process all controlling-expression values that are less than 6—that is, all failing grades. If no match occurs and the switch does not contain a default case, program control simply continues with the first statement after the switch.

**Error-Prevention Tip 4.7**

*In a switch statement, ensure that you test for all possible values of the controlling expression.*

### *Displaying the Grade Report*

Lines 59–80 output a report based on the grades entered (as shown in the input/output window in Fig. 4.9). Line 62 determines whether the user entered at least one grade—this helps us avoid dividing by zero. If so, line 65 calculates the average of the grades. Lines 68–77 then output the total of all the grades, the class average and the number of students who received each letter grade. If no grades were entered, line 80 outputs an appropriate message. The output in Fig. 4.9 shows a sample grade report based on 10 grades.

### *switch Statement UML Activity Diagram*

Figure 4.10 shows the UML activity diagram for the general switch statement. Most switch statements use a break in each case to terminate the switch statement after processing the case. Figure 4.10 emphasizes this by including break statements in the activity diagram. The diagram makes it clear that the break statement at the end of a case causes control to exit the switch statement immediately.

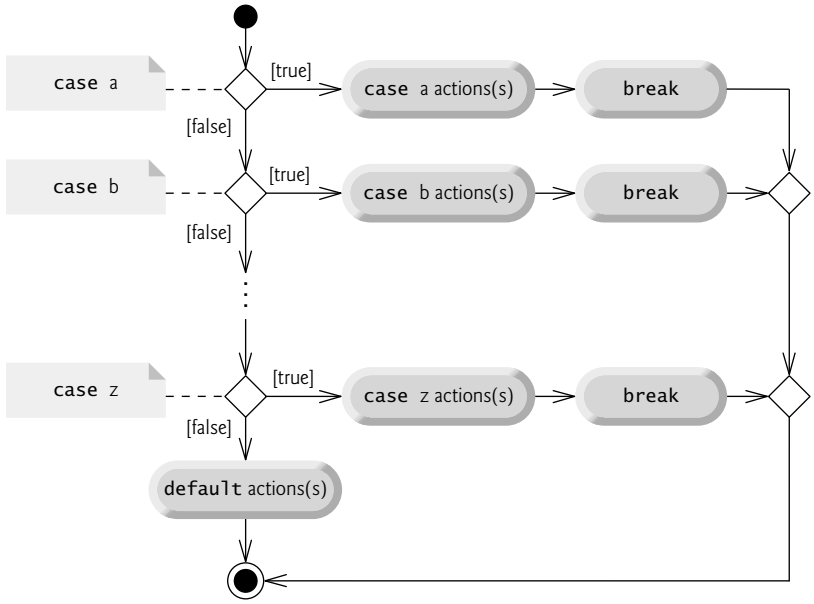


**Fig. 4.10** | `switch` multiple-selection statement UML activity diagram with `break` statements.

The `break` statement is *not* required for the `switch`'s last `case` (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`.

### Error-Prevention Tip 4.8

*Provide a `default` case in `switch` statements. This focuses you on the need to process exceptional conditions.*

### Good Programming Practice 4.3

*Although each `case` and the `default` case in a `switch` can occur in any order, place the `default` case last. When the `default` case is listed last, the `break` for that case is not required.*

### *Notes on the Expression in Each `case` of a `switch`*

When using the `switch` statement, remember that each `case` must contain a `String` ot a constant integral expression—that is, any combination of integer constants that evaluates to a constant integer value (e.g., –7, 0 or 221). An integer constant is simply an integer value. In addition, you can use **character constants**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`—which represent the integer values of characters and `enum` constants (introduced in Section 5.10). (Appendix B shows the integer values of the characters in the ASCII character set, which is a subset of the Unicode® character set used by Java.)

The expression in each `case` can also be a **constant variable**—a variable containing a value which does not change for the entire program. Such a variable is declared with keyword `final` (discussed in Chapter 5). Java has a feature called `enum` types, which we also present in Chapter 5—`enum` type constants can also be used in `case` labels.

In Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces, 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.

## 4.7 break and continue Statements

In addition to selection and repetition statements, Java provides statements `break` (which we discussed in the context of the `switch` statement)and **`continue`** (presented in this section and online Appendix L) to alter the flow of control. The preceding section showed how `break` can be used to terminate a `switch` statement's execution. This section discusses how to use `break` in repetition statements.

### *break Statement*

The `break` statement, when executed in a `while`, `for`, `do...while` or `switch`, causes *immediate* exit from that statement. Execution continues with the first statement after the control statement. Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` (as in Fig. 4.9). Figure 4.11 demonstrates a `break` statement exiting a `for`.

```
1  // Fig. 4.11: BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest
4  {
5     public static void main(String[] args)
6     {
7        int count; // control variable also used after loop terminates
8
9        for (count = 1; count <= 10; count++) // loop 10 times
10       {
11          if (count == 5)
12             break; // terminates loop if count is 5
13
14          System.out.printf("%d ", count);
15       }
16
17       System.out.printf("%nBroke out of loop at count = %d%n", count);
18    }
19 } // end class BreakTest
```

```
1 2 3 4
Broke out of loop at count = 5
```

**Fig. 4.11** | `break` statement exiting a `for` statement.

When the `if` statement nested at lines 11–12 in the `for` statement (lines 9–15) detects that `count` is `5`, the `break` statement at line 12 executes. This terminates the `for` statement, and the program proceeds to line 17 (immediately after the `for` statement), which displays a message indicating the value of the control variable when the loop terminated. The loop fully executes its body only four times instead of 10.

***continue* Statement**

The continue statement, when executed in a while, for or do...while, skips the remaining statements in the loop body and proceeds with the *next iteration* of the loop. In while and do...while statements, the program evaluates the loop-continuation test immediately after the continue statement executes. In a for statement, the increment expression executes, then the program evaluates the loop-continuation test.

```
1  // Fig. 4.12: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest
4  {
5     public static void main(String[] args)
6     {
7        for (int count = 1; count <= 10; count++) // loop 10 times
8        {
9           if (count == 5)
10             continue; // skip remaining code in loop body if count is 5
11
12          System.out.printf("%d ", count);
13       }
14
15       System.out.printf("%nUsed continue to skip printing 5%n");
16    }
17 } // end class ContinueTest
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

**Fig. 4.12** | continue statement terminating an iteration of a for statement.

Figure 4.12 uses continue (line 10) to skip the statement at line 12 when the nested if determines that count's value is 5. When the continue statement executes, program control continues with the increment of the control variable in the for statement (line 7).

In Section 4.3, we stated that while could be used in most cases in place of for. This is *not* true when the increment expression in the while follows a continue statement. In this case, the increment does *not* execute before the program evaluates the repetition-continuation condition, so the while does not execute in the same manner as the for.

**Software Engineering Observation 4.2**

*Some programmers feel that* break *and* continue *violate structured programming. Since the same effects are achievable with structured programming techniques, these programmers do not use* break *or* continue.

**Software Engineering Observation 4.3**

*There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes 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.*

## 4.8 Logical Operators

The `if`, `if...else`, `while`, `do...while` and `for` statements each require a *condition* to determine how to continue a program's flow of control. So far, we've studied only simple conditions, such as `count <= 10`, `number != sentinelValue` and `total > 1000`. Simple conditions are expressed in terms of the relational operators >, <, >= and <= and the equality operators == and !=, and each expression tests only 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` statements. Sometimes control statements require more complex conditions to determine a program's flow of control.

Java's **logical operators** enable you to form more complex conditions by *combining* simple conditions. The logical operators are && (conditional AND), || (conditional OR), & (boolean logical AND), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR) and ! (logical NOT). [*Note:* The &, | and ^ operators are also bitwise operators when they're applied to integral operands. We discuss the bitwise operators in online Appendix K.]

### *Conditional AND (&&) Operator*

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 **&&** (**conditional AND**) operator, as follows:

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

This `if` statement contains two simple conditions. The condition `gender == FEMALE` compares variable `gender` to the constant `FEMALE` to determine whether a person is female. The condition `age >= 65` might be evaluated to determine whether a person is a senior citizen. The `if` statement considers the combined condition

```
gender == FEMALE && age >= 65
```

which is true if and only if *both* simple conditions are true. In this case, the `if` statement's body increments `seniorFemales` by 1. If either or both of the simple conditions are false, the program skips the increment. Some programmers find that the preceding combined condition is more readable when *redundant* parentheses are added, as in:

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

The table in Fig. 4.13 summarizes the && operator. The table shows all four possible combinations of `false` and `true` values for *expression1* and *expression2*. Such tables are called **truth tables**. Java evaluates to `false` or `true` all expressions that include relational operators, equality operators or logical operators.

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

**Fig. 4.13** | && (conditional AND) operator truth table.

### *Conditional OR (||) Operator*

Now 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 `||` (**conditional OR**) operator, as in the following program segment:

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

The preceding statement also contains two simple conditions. The condition `semesterAverage >= 90` evaluates to determine whether the student deserves an A in the course because of a solid performance throughout the semester. The condition `finalExam >= 90` evaluates to determine whether 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

```
(semesterAverage >= 90) || (finalExam >= 90)
```

and awards the student an A if *either or both* of the simple conditions are true. The only time the message `"Student grade is A"` is *not* printed is when *both* of the simple conditions are *false*. Figure 4.14 shows the truth table for operator conditional OR (`||`). Operator `&&` has a higher precedence than operator `||`. Both operators associate from left to right.

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

**Fig. 4.14** | `||` (conditional OR) operator truth table.

### *Short-Circuit Evaluation of Complex Conditions*

The parts of an expression containing `&&` or `||` operators are evaluated *only* until it's known whether the condition is true or false. Thus, evaluation of the expression

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

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

**Common Programming Error 4.8**

*In expressions using operator `&&`, a condition—we'll call this the* dependent condition*—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed* after *the `&&` operator to prevent errors. Consider the expression `(i != 0) && (10 / i == 2)`. The dependent condition `(10 / i == 2)` must appear after the `&&` operator to prevent the possibility of division by zero.*

### *Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators*

The **boolean logical AND** (**&**) and **boolean logical inclusive OR** (**|**) operators are identical to the **&&** and **||** operators, except that the **&** and **|** operators *always* evaluate *both* of their operands (i.e., they do *not* perform short-circuit evaluation). So, the expression

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

evaluates `age >= 65` *regardless* of whether `gender` is equal to `1`. This is useful if the right operand has a required **side effect**—a modification of a variable's value. For example, the expression

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

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

**Error-Prevention Tip 4.9**

*For clarity, avoid expressions with side effects (such as assignments) in conditions. They can make code harder to understand and can lead to subtle logic errors.*

**Error-Prevention Tip 4.10**

*Assignment (=) expressions generally should* not *be used in conditions. Every condition* must *result in a* `boolean` *value; otherwise, a compilation error occurs. In a condition, an assignment will compile* only *if a* `boolean` *expression is assigned to a* `boolean` *variable.*

### *Boolean Logical Exclusive OR (^)*

A simple condition containing the **boolean logical exclusive OR** (**^**) operator is `true` *if and only if one of its operands is* `true` *and the other is* `false`. If both are `true` or both are `false`, the entire condition is `false`. Figure 4.15 is a truth table for the boolean logical exclusive OR operator (^). This operator is guaranteed to evaluate *both* of its operands.

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

**Fig. 4.15** | ^ (boolean logical exclusive OR) operator truth table.

### *Logical Negation (!) Operator*

The **!** (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition. Unlike the logical operators **&&**, **||**, **&**, **|** and **^**, which are ***binary*** operators that combine two conditions, the logical negation operator is a *unary* operator that has only one condition as an operand. The operator is placed *before* a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, as in the program segment

```
if (! (grade == sentinelValue))
   System.out.printf("The next grade is %d%n", grade);
```

which executes the `printf` call only if `grade` is *not* equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a *higher* precedence than the equality operator.

In most cases, you 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:

```
if (grade != sentinelValue)
   System.out.printf("The next grade is %d%n", grade);
```

This flexibility can help you express a condition in a more convenient manner. Figure 4.16 is a truth table for the logical negation operator.

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

**Fig. 4.16** | `!` (logical NOT) operator truth table.

*Logical Operators Example*

Figure 4.17 uses logical operators to produce the truth tables discussed in this section. The output shows the `boolean` expression that was evaluated and its result. We used the **`%b` format specifier** to display the word "true" or the word "false" based on a `boolean` expression's value. Lines 9–13 produce the truth table for `&&`. Lines 16–20 produce the truth table for `||`. Lines 23–27 produce the truth table for `&`. Lines 30–35 produce the truth table for `|`. Lines 38–43 produce the truth table for `^`. Lines 46–47 produce the truth table for `!`.

```
1  // Fig. 4.17: LogicalOperators.java
2  // Logical operators.
3
4  public class LogicalOperators
5  {
6     public static void main(String[] args)
7     {
8        // create truth table for && (conditional AND) operator
9        System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
10          "Conditional AND (&&)", "false && false", (false && false)  ,
11          "false && true", (false && true)  ,
12          "true && false", (true && false)  ,
13          "true && true", (true && true)  );
14
15       // create truth table for || (conditional OR) operator
16       System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
17          "Conditional OR (||)", "false || false", (false || false)  ,
18          "false || true", (false || true)  ,
19          "true || false", (true || false)  ,
20          "true || true", (true || true)  );
```

**Fig. 4.17** | Logical operators. (Part 1 of 3.)

```
21
22        // create truth table for & (boolean logical AND) operator
23        System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
24           "Boolean logical AND (&)", "false & false", (false & false)  ,
25           "false & true", (false & true)  ,
26           "true & false", (true & false)  ,
27           "true & true", (true & true)  );
28
29        // create truth table for | (boolean logical inclusive OR) operator
30        System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
31           "Boolean logical inclusive OR (|)",
32           "false | false", (false | false)  ,
33           "false | true", (false | true)  ,
34           "true | false", (true | false)  ,
35           "true | true", (true | true)  );
36
37        // create truth table for ^ (boolean logical exclusive OR) operator
38        System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%n",
39           "Boolean logical exclusive OR (^)",
40           "false ^ false", (false ^ false)  ,
41           "false ^ true", (false ^ true)  ,
42           "true ^ false", (true ^ false)  ,
43           "true ^ true", (true ^ true)  );
44
45        // create truth table for ! (logical negation) operator
46        System.out.printf("%s%n%s: %b%n%s: %b%n", "Logical NOT (!)",
47           "!false", (!false)  , "!true", (!true)  );
48     }
49  } // end class LogicalOperators
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true
```

**Fig. 4.17** | Logical operators. (Part 2 of 3.)

```
Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

**Fig. 4.17** | Logical operators. (Part 3 of 3.)

***Precedence and Associativity of the Operators Presented So Far***
Figure 4.18 shows the precedence and associativity of the Java operators introduced so far. The operators are shown from top to bottom in decreasing order of precedence.

| Operators | Associativity | Type |
|---|---|---|
| `++ --` | right to left | unary postfix |
| `++ -- + - ! (type)` | right to left | unary prefix |
| `* / %` | 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 | conditional AND |
| `\|\|` | left to right | conditional OR |
| `?:` | right to left | conditional |
| `= += -= *= /= %=` | right to left | assignment |

**Fig. 4.18** | Precedence/associativity of the operators discussed so far.

## 4.9 Structured Programming Summary

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

***Java Control Statements Are Single-Entry/Single-Exit***
Figure 4.19 uses UML activity diagrams to summarize Java's control statements. The initial and final states indicate the *single entry point* and the *single exit point* of each control

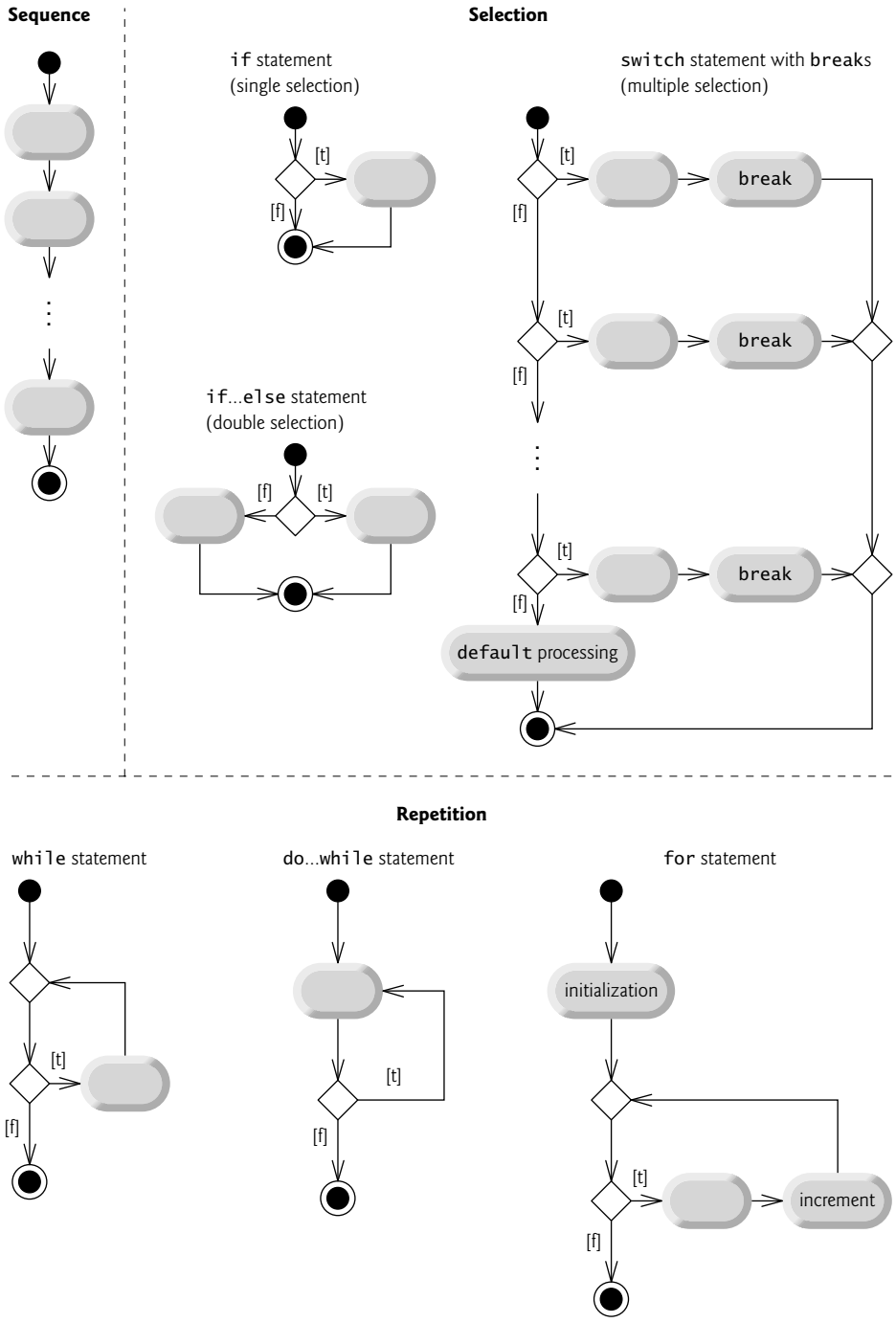


**Fig. 4.19** | Java's single-entry/single-exit sequence, selection and repetition statements.

statement. Arbitrarily connecting individual symbols in an activity diagram can lead to unstructured programs. Therefore, the programming profession has chosen a limited set of control statements that can be combined in only two simple ways to build structured programs.

For simplicity, Java includes only *single-entry/single-exit* control statements—there's only one way to enter and only one way to exit each control statement. Connecting control statements in sequence to form structured programs is simple. The final state of one control statement is connected to the initial state of the next—that is, the control statements are placed one after another in a program in sequence. We call this *control-statement stacking*. The rules for forming structured programs also allow for control statements to be *nested*.

### *Rules for Forming Structured Programs*

Figure 4.20 shows the rules for forming structured programs. The rules assume that action states may be used to indicate *any* action. The rules also assume that we begin with the simplest activity diagram (Fig. 4.21) consisting of only an initial state, an action state, a final state and transition arrows.

**Rules for forming structured programs**

1. Begin with the simplest activity diagram (Fig. 4.21).
2. Any action state can be replaced by two action states in sequence.
3. Any action state can be replaced by any control statement (sequence of action states, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4. Rules 2 and 3 can be applied as often as you like and in any order.

**Fig. 4.20** | Rules for forming structured programs.

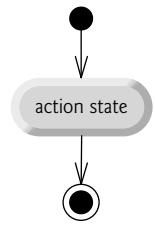


**Fig. 4.21** | Simplest activity diagram.

Applying the rules in Fig. 4.20 always results in a properly structured activity diagram with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest activity diagram results in an activity diagram containing many action states in sequence (Fig. 4.22). Rule 2 generates a *stack* of control statements, so let's call rule 2 the **stacking rule**. The vertical dashed lines in Fig. 4.22 are not part of the UML—we use them to separate the four activity diagrams that demonstrate rule 2 of Fig. 4.20 being applied.

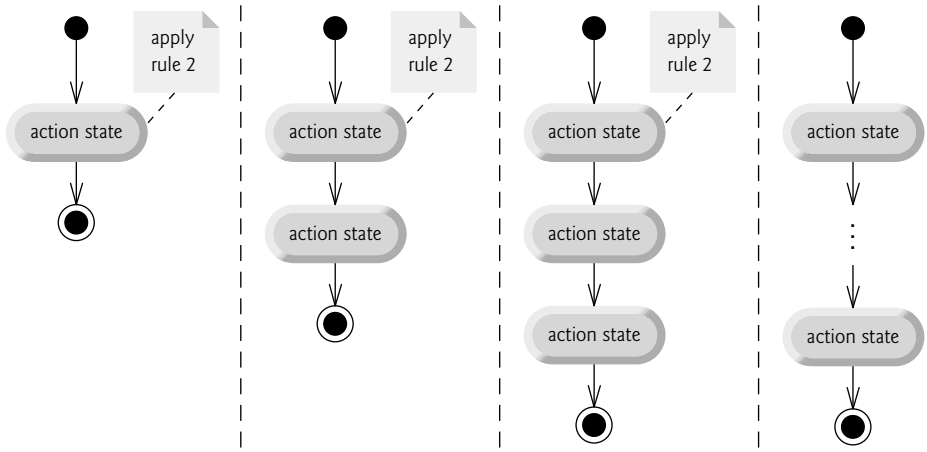


**Fig. 4.22** | Repeatedly applying rule 2 of Fig. 4.20 to the simplest activity diagram.

Rule 3 is called the **nesting rule**. Repeatedly applying rule 3 to the simplest activity diagram results in one with neatly *nested* control statements. For example, in Fig. 4.23, the action state in the simplest activity diagram is replaced with a double-selection (`if...else`) statement. Then rule 3 is applied again to the action states in the double-selection statement, replacing each with a double-selection statement. The dashed action-state symbol around each double-selection statement represents the action state that was replaced. [*Note:* The dashed arrows and dashed action-state symbols shown in Fig. 4.23 are not part of the UML. They're used here to illustrate that *any* action state can be replaced with a control statement.]

Rule 4 generates larger, more involved and more deeply nested statements. The diagrams that emerge from applying the rules in Fig. 4.20 constitute the set of all possible structured activity diagrams 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 control statements and assemble them in *only two* simple ways.

If the rules in Fig. 4.20 are followed, an "unstructured' activity diagram (like the one in Fig. 4.24) cannot be created. If you're uncertain about whether a particular diagram is structured, apply the rules of Fig. 4.20 in reverse to reduce it to the simplest activity diagram. If you can reduce it, the original diagram is structured; otherwise, it's not.

### *Three Forms of Control*

Structured programming promotes simplicity. Only three forms of control are needed to implement an algorithm:

- sequence
- selection
- repetition

The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute. Selection is implemented in one of three ways:

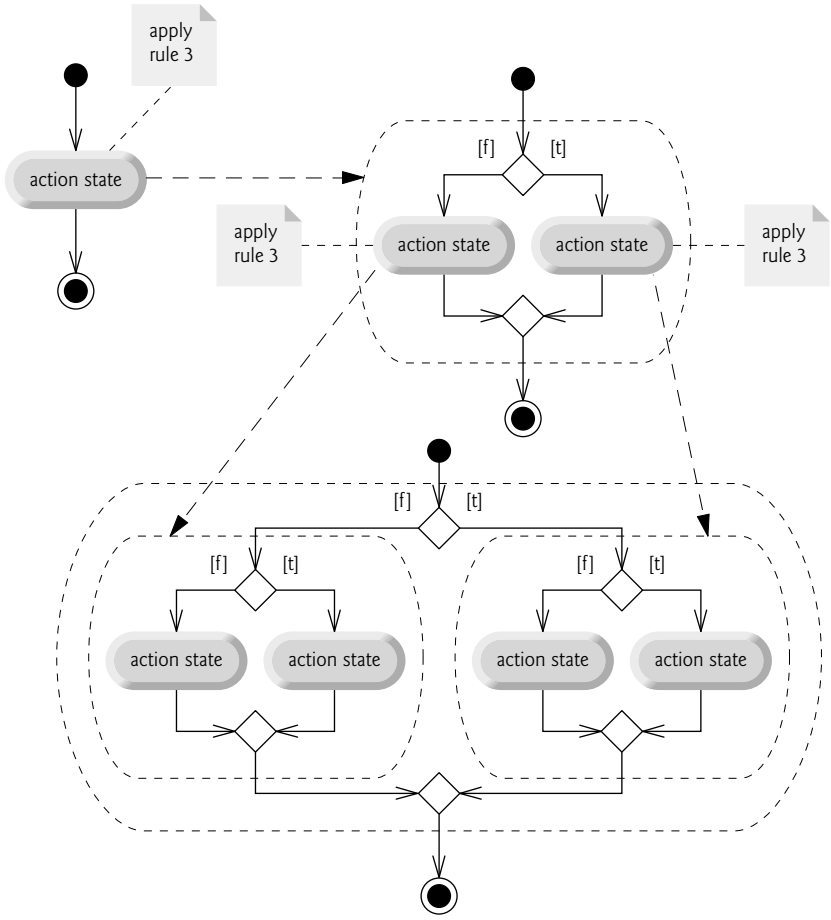


**Fig. 4.23** | Repeatedly applying rule 3 of Fig. 4.20 to the simplest activity diagram.

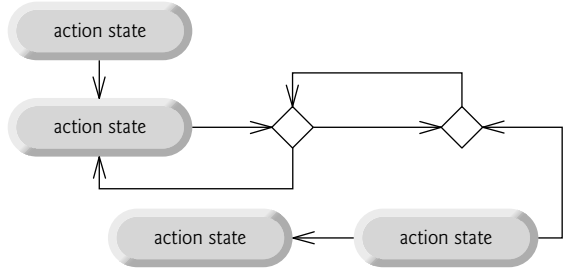


**Fig. 4.24** | "Unstructured" activity diagram.

- `if` statement (single selection)
- `if...else` statement (double selection)
- `switch` statement (multiple selection)

In fact, it's straightforward to prove that the simple `if` statement is sufficient to provide *any* form of selection—everything that can be done with the `if...else` statement and the `switch` statement can be implemented by combining `if` statements (although perhaps not as clearly and efficiently).

Repetition is implemented in one of three ways:

- `while` statement
- `do...while` statement
- `for` statement

[*Note:* There's a fourth repetition statement—the *enhanced `for` statement*—that we discuss in Section 6.6.] It's straightforward to prove that the `while` statement is sufficient to provide *any* form of repetition. Everything that can be done with `do...while` and `for` can be done with the `while` statement (although perhaps not as conveniently).

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

- sequence
- `if` statement (selection)
- `while` statement (repetition)

and that these can be combined in only two ways—*stacking* and *nesting*. Indeed, structured programming is the essence of simplicity.

## 4.10 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals

Our next program (Fig. 4.25) demonstrates drawing rectangles and ovals, using the `Graphics` methods **`drawRect`** and **`drawOval`**, respectively. Method `paintComponent` (lines 10–21) performs the actual drawing. Remember, the first statement in every `paintComponent` method must be a call to `super.paintComponent`, as in line 12. Lines 14–20 loop 10 times to draw 10 rectangles and 10 ovals.

```
1  // Fig. 4.25: Shapes.java
2  // Drawing a cascade of shapes based on the user's choice.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5  import javax.swing.JFrame;
6
7  public class Shapes extends JPanel
8  {
```

**Fig. 4.25** | Drawing a cascade of shapes based on the user's choice. (Part 1 of 2.)

```
 9     // draws a cascade of shapes starting from the top-left corner
10     public void paintComponent(Graphics g)
11     {
12        super.paintComponent(g);
13
14        for (int i = 0; i < 10; i++)
15        {
16           g.drawRect(10 + i * 10, 10 + i * 10,
17              50 + i * 10, 50 + i * 10);
18           g.drawOval(240 + i * 10, 10 + i * 10,
19              50 + i * 10, 50 + i * 10);
20        }
21     }
22
23     public static void main( String[] args )
24     {
25        Shapes panel = new Shapes(); // create the panel
26        JFrame application = new JFrame(); // creates a new JFrame
27
28        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29        application.add(panel);
30        application.setSize(500, 290);
31        application.setVisible(true);
32     }
33  } // end class Shapes
```

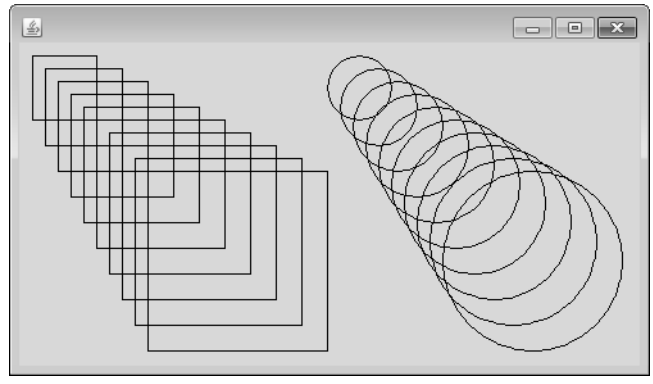

**Fig. 4.25** | Drawing a cascade of shapes based on the user's choice. (Part 2 of 2.)

Lines 16–17 call `Graphics` method `drawRect`, which requires four arguments. The first two represent the *x*- and *y*-coordinates of the upper-left corner of the rectangle; the next two represent the rectangle's width and height. In this example, we start at a position 10 pixels down and 10 pixels right of the top-left corner, and every iteration of the loop moves the upper-left corner another 10 pixels down and to the right. The width and the height of the rectangle start at 50 pixels and increase by 10 pixels in each iteration.

Lines 18–19 in the loop draw ovals. Method `drawOval` creates an imaginary rectangle called a **bounding rectangle** and places inside it an oval that touches the midpoints of all four sides. The method's four arguments represent the *x*- and *y*-coordinates of the upper-

left corner of the bounding rectangle and the bounding rectangle's width and height. The values passed to `drawOval` in this example are exactly the same as those passed to `drawRect` in lines 16–17, except that the first oval's bounding box has an *x*-coordinate that starts 240 pixels from the left side of the panel. Since the width and height of the bounding rectangle are identical in this example, lines 18–19 draw a circle. As an exercise, modify the program to draw both rectangles and ovals with the same arguments. This will allow you to see how each oval touches all four sides of the corresponding rectangle.

Line 25 in `main` creates a `Shapes` object. Lines 26–31 perform the standard operations that create and set up a window in this case study—create a frame, set it to exit the application when closed, add the drawing to the frame, set the frame size and make it visible.

### *GUI and Graphics Case Study Exercise*

**4.1** Draw 12 concentric circles in the center of a `JPanel` (Fig. 4.26). The innermost circle should have a radius of 10 pixels, and each successive circle should have a radius 10 pixels larger than the previous one. Begin by finding the center of the `JPanel`. To get the upper-left corner of a circle, move up one radius and to the left one radius from the center. The width and height of the bounding rectangle are both the same as the circle's diameter (i.e., twice the radius).

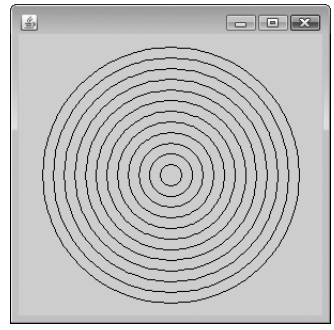

**Fig. 4.26** | Drawing concentric circles.

## 4.11 Wrap-Up

In this chapter, we completed our introduction to control statements, which enable you to control the flow of execution in methods. Chapter 3 discussed `if`, `if...else` and `while`. This chapter demonstrated `for`, `do...while` and `switch`. We showed that any algorithm can be developed using combinations of the sequence structure, the three types of selection statements—`if`, `if...else` and `switch`—and the three types of repetition statements—`while`, `do...while` and `for`. In this chapter and Chapter 3, we discussed how you can combine these building blocks to utilize proven program-construction and problem-solving techniques. You used the `break` statement to exit a `switch` statement and to immediately terminate a loop, and used a `continue` statement to terminate a loop's current iteration and proceed with the loop's next iteration. This chapter also introduced Java's logical operators, which enable you to use more complex conditional expressions in control statements. In Chapter 5, we examine methods.

## Summary

### *Section 4.2 Essentials of Counter-Controlled Repetition*

- Counter-controlled repetition (p. 122) requires a control variable, the initial value of the control variable, the increment by which the control variable is modified each time through the loop (also known as each iteration of the loop) and the loop-continuation condition that determines whether looping should continue.
- You can declare a variable and initialize it in the same statement.

### *Section 4.3* `for` *Repetition Statement*

- The `while` statement can be used to implement any counter-controlled loop.
- The `for` statement (p. 124) specifies all the details of counter-controlled repetition in its header
- When the `for` statement begins executing, its control variable is declared and initialized. If the loop-continuation condition is initially true, the body executes. After executing the loop's body, the increment expression executes. Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop.
- The general format of the `for` statement is

```
for (initialization; loopContinuationCondition; increment)
   statement
```

  where the *initialization* expression names the loop's control variable and provides its initial value, *loopContinuationCondition* determines whether the loop should continue executing and *increment* modifies the control variable's value, so that the loop-continuation condition eventually becomes false. The two semicolons in the `for` header are required.
- Most `for` statements can be represented with equivalent `while` statements as follows:

```
initialization;
while (loopContinuationCondition)
{
   statement
   increment;
}
```

- Typically, `for` statements are used for counter-controlled repetition and `while` statements for sentinel-controlled repetition.
- If the *initialization* expression in the `for` header declares the control variable, the control variable can be used only in that `for` statement—it will not exist outside the `for` statement.
- The expressions in a `for` header are optional. If the *loopContinuationCondition* is omitted, Java assumes that it's always true, thus creating an infinite loop. You might omit the *initialization* expression if the control variable is initialized before the loop. You might omit the *increment* expression if the increment is calculated with statements in the loop's body or if no increment is needed.
- The increment expression in a `for` acts as if it's a standalone statement at the end of the `for`'s body.
- A `for` statement can count downward by using a negative increment—i.e., a decrement (p. 127).
- If the loop-continuation condition is initially `false`, the `for` statement's body does not execute.

### *Section 4.4 Examples Using the* `for` *Statement*

- Java treats floating-point constants like `1000.0` and `0.05` as type `double`. Similarly, Java treats whole-number constants like `7` and `-22` as type `int`.
- The format specifier `%4s` outputs a `String` in a field width (p. 130) of 4—that is, `printf` displays the value with at least 4 character positions. If the value to be output is less than 4 character po-

sitions wide, the value is right justified (p. 130) in the field by default. If the value is greater than 4 character positions wide, the field width expands to accommodate the appropriate number of characters. To left justify (p. 130) the value, use a negative integer to specify the field width.

- `Math.pow(x, y)` (p. 131) calculates the value of *x* raised to the yth power. The method receives two `double` arguments and returns a `double` value.
- The comma (,) formatting flag (p. 131) in a format specifier indicates that a floating-point value should be output with a grouping separator (p. 131). The actual separator used is specific to the user's locale (i.e., country). In the United States, the number will have commas separating every three digits and a decimal point separating the fractional part of the number, as in 1,234.45.
- The . in a format specifier indicates that the integer to its right is the number's precision.

### *Section 4.5 `do...while` Repetition Statement*

- The `do...while` statement (p. 132) is similar to the `while` statement. In the `while`, the program tests the loop-continuation condition at the beginning of the loop, before executing its body; if the condition is false, the body never executes. The `do...while` statement tests the loop-continuation condition *after* executing the loop's body; therefore, the body always executes at least once.

### *Section 4.6 `switch` Multiple-Selection Statement*

- The `switch` statement (p. 134) performs different actions based on the possible values of a constant integral expression (a constant value of type `byte`, `short`, `int` or `char`, but not `long`), or a `String`.
- The end-of-file indicator is a system-dependent keystroke combination that terminates user input. On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence <*Ctrl*> *d* on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key. On Windows systems, enter end-of-file by typing <*Ctrl*> *z*.
- `Scanner` method `hasNext` (p. 137) determines whether there's more data to input. This method returns the `boolean` value `true` if there's more data; otherwise, it returns `false`. As long as the end-of-file indicator has not been typed, method `hasNext` will return `true`.
- The `switch` statement consists of a block that contains a sequence of `case` labels (p. 137) and an optional `default` case (p. 137).
- In a `switch`, the program evaluates the controlling expression and compares its value with each `case` label. If a match occurs, the program executes the statements for that `case`.
- Listing cases consecutively with no statements between them enables the cases to perform the same set of statements.
- Every value you wish to test in a `switch` must be listed in a separate `case` label.
- Each `case` can have multiple statements, and these need not be placed in braces.
- A `case`'s statements typically end with a `break` statement (p. 137) that terminates the `switch`'s execution.
- Without `break` statements, each time a match occurs in the `switch`, the statements for that case and subsequent cases execute until a `break` statement or the end of the `switch` is encountered.
- If no match occurs between the controlling expression's value and a `case` label, the optional `default` case executes. If no match occurs and the `switch` does not contain a `default` case, program control simply continues with the first statement after the `switch`.

### *Section 4.7 `break` and `continue` Statements*

- The `break` statement, when executed in a `while`, `for`, `do...while` or `switch`, causes immediate exit from that statement.

- The `continue` statement (p. 140), when executed in a `while`, `for` or `do...while`, skips the loop's remaining body statements and proceeds with its next iteration. In `while` and `do...while` statements, the program evaluates the loop-continuation test immediately. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

### *Section 4.8 Logical Operators*

- Simple conditions are expressed in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`, and each expression tests only one condition.
- Logical operators (p. 142) enable you to form more complex conditions by combining simple conditions. The logical operators are `&&` (conditional AND), `||` (conditional OR), `&` (boolean logical AND), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR) and `!` (logical NOT).
- To ensure that two conditions are *both* true, use the `&&` (conditional AND) operator. If either or both of the simple conditions are false, the entire expression is false.
- To ensure that either *or* both of two conditions are true, use the `||` (conditional OR) operator, which evaluates to true if either or both of its simple conditions are true.
- A condition using `&&` or `||` operators (p. 142) uses short-circuit evaluation (p. 143)—they're evaluated only until it's known whether the condition is true or false.
- The `&` and `|` operators (p. 144) work identically to the `&&` and `||` operators but always evaluate both operands.
- A simple condition containing the boolean logical exclusive OR (`^`; p. 144) operator is `true` *if and only if one of its operands is* `true` *and the other is* `false`. If both operands are `true` or both are `false`, the entire condition is `false`. This operator is also guaranteed to evaluate both of its operands.
- The unary `!` (logical NOT; p. 144) operator "reverses" the value of a condition.

## Self-Review Exercises

**4.1** Fill in the blanks in each of the following statements:

a) Typically, _____ statements are used for counter-controlled repetition and _____ statements for sentinel-controlled repetition.

b) The `do...while` statement tests the loop-continuation condition _____ executing the loop's body; therefore, the body always executes at least once.

c) The _____ statement selects among multiple actions based on the possible values of an integer variable or expression, or a `String`.

d) The _____ statement, when executed in a repetition statement, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.

e) The _____ operator can be used to ensure that two conditions are *both* true before choosing a certain path of execution.

f) If the loop-continuation condition in a `for` header is initially _____, the program does not execute the `for` statement's body.

g) Methods that perform common tasks and do not require objects are called _____ methods.

**4.2** State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `default` case is required in the `switch` selection statement.

b) The `break` statement is required in the last case of a `switch` selection statement.

c) The expression `((x > y) && (a < b))` is true if either `x > y` is true or `a < b` is true.

d) An expression containing the `||` operator is true if either or both of its operands are true.

e) The comma (`,`) formatting flag in a format specifier (e.g., `%,20.2f`) indicates that a value should be output with a thousands separator.

f) To test for a range of values in a `switch` statement, use a hyphen (-) between the start and end values of the range in a `case` label.
g) Listing cases consecutively with no statements between them enables the cases to perform the same set of statements.

**4.3** Write a Java statement or a set of Java statements to accomplish each of the following tasks:
a) Sum the odd integers between 1 and 99, using a `for` statement. Assume that the integer variables `sum` and `count` have been declared.
b) Calculate the value of `2.5` raised to the power of `3`, using the `pow` method.
c) Print the integers from 1 to 20, using a `while` loop and the counter variable `i`. Assume that the variable `i` has been declared, but not initialized. Print only five integers per line. [*Hint:* Use the calculation `i % 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 part (c), using a `for` statement.

**4.4** Find the error in each of the following code segments, and explain how to correct it:
a)
```
i = 1;
while (i <= 10);
   ++i;
}
```
b)
```
for (k = 0.1; k != 1.0; k += 0.1)
   System.out.println(k);
```
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

**4.1** a) `for`, `while`. b) after. c) `switch`. d) `continue`. e) `&&` (conditional AND). f) `false`. g) `static`.

**4.2** a) False. The `default` case is optional. If no default action is needed, then there's no need for a `default` case. b) False. The `break` statement is used to exit the `switch` statement. The `break` statement is not required for the last case in a `switch` statement. c) False. *Both* of the relational expressions must be true for the entire expression to be true when using the `&&` operator. d) True. e) True. f) False. The `switch` statement does not provide a mechanism for testing ranges of values, so every value that must be tested should be listed in a separate `case` label. g) True.

**4.3** a)
```
sum = 0;
for (count = 1; count <= 99; count += 2)
   sum += count;
```

b)
```
double result = Math.pow(2.5, 3);
```
c)
```
i = 1;

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

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

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

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

**4.4** a) Error: The semicolon after the `while` header causes an infinite loop, and there's a missing left brace.
Correction: Replace the semicolon by a `{`, or remove both the `;` and the `}`.

b) Error: Using a floating-point number to control a `for` statement 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 (k = 1; k != 10; k++)
   System.out.println((double) k / 10);
```

c) Error: The missing code is the `break` statement in the statements for the first `case`.
Correction: Add a `break` statement at the end of the statements for the first `case`. This omission is not necessarily an error if you want the statement of `case 2:` to execute every time the `case 1:` statement executes.

d) Error: An improper relational operator is used in the `while`'s continuation condition.
Correction: Use `<=` rather than `<`, or change `10` to `11`.

## Exercises

**4.5** Describe the four basic elements of counter-controlled repetition.

**4.6** Compare and contrast the `while` and `for` repetition statements.

**4.7** Discuss a situation in which it would be more appropriate to use a `do...while` statement than a `while` statement. Explain why.

**4.8** Compare and contrast the `break` and `continue` statements.

**4.9** Find and correct the error(s) in each of the following segments of code:

a)
```
For (i = 100, i >= 1, i++)
   System.out.println(i);
```

b) 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");
}
```

c) The following code should output the odd integers from 19 to 1:

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

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);
```

**4.10** What does the following program do?

```
1  // Exercise 4.10: Printing.java
2  public class Printing
3  {
4     public static void main(String[] args)
5     {
6        for (int i = 1; i <= 10; i++)
7        {
8           for (int j = 1; j <= 5; j++)
9              System.out.print('@');
10
11          System.out.println();
12       }
13    }
14 } // end class Printing
```

**4.11** ***(Find the Smallest Value)*** 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.

**4.12** ***(Calculating the Product of Odd Integers)*** Write an application that calculates the product of the odd integers from 1 to 15.

**4.13** ***(Factorials)*** *Factorials* are 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 calculates the factorials of 1 through 20. Use type `long`. Display the results in tabular format. What difficulty might prevent you from calculating the factorial of 100?

**4.14** ***(Modified Compound-Interest Program)*** Modify the compound-interest application of Fig. 4.6 to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9% and 10%. Use a `for` loop to vary the interest rate.

**4.15** ***(Triangle Printing Program)*** Write an application that displays the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (`*`) should be print-

ed by a single statement of the form `System.out.print('*');` which causes the asterisks to print side by side. A statement of the form `System.out.println();` can be used to move 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.]

```
(a)          (b)          (c)          (d)
*            **********   **********            *
**           *********     *********           **
***          ********       ********         ***
****         *******         *******       ****
*****        ******           ******     *****
******       *****             *****   ******
*******      ****               **** *******
********     ***                 *** ********
*********    **                   ** *********
**********   *                     * **********
```

**4.16** ***(Bar Chart Printing Program)*** One interesting application of computers is to display graphs and bar charts. Write an application that reads five numbers between 1 and 30. For each number that's read, your program should display the same number of adjacent asterisks. For example, if your program reads the number 7, it should display *******. Display the bars of asterisks *after* you read all five numbers.

**4.17** ***(Calculating Sales)*** An online retailer sells five products whose retail prices are as follows: Product 1, $2.98; product 2, $4.50; product 3, $9.98; product 4, $4.49 and product 5, $6.87. Write an application that reads a series of pairs of numbers as follows:

a) product number
b) quantity sold

Your program should use a `switch` statement to determine the retail price for each product. It should calculate and display the total retail value of all products sold. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

**4.18** ***(Modified Compound-Interest Program)*** Modify the application in Fig. 4.6 to use only integers to calculate the compound interest. [*Hint:* Treat all monetary amounts as integral numbers of pennies. Then break the result into its dollars and cents portions by using the division and remainder operations, respectively. Insert a period between the dollars and the cents portions.]

**4.19** Assume that `i = 1`, `j = 2`, `k = 3` and `m = 2`. What does each of the following statements print?

a) `System.out.println(i == 1);`
b) `System.out.println(j == 3);`
c) `System.out.println((i >= 1) && (j < 4));`
d) `System.out.println((m <= 99) & (k < m));`
e) `System.out.println((j >= i) || (k == m));`
f) `System.out.println((k + m < j) | (3 - j >= k));`
g) `System.out.println(!(k > m));`

**4.20** ***(Calculating the Value of π)*** Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of π approximated by computing the first 200,000 terms of this series. How many terms do you have to use before you first get a value that begins with 3.14159?

**4.21** ***(Pythagorean Triples)*** A right triangle can have sides whose lengths are all integers. The set of three integer values for the lengths of the sides of a right triangle is called a Pythagorean triple. The lengths of the three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Write an application that displays a table of the Pythagorean triples for `side1`, `side2` and the `hypotenuse`, all no larger than 500. Use a triple-nested `for` loop that tries all possibilities. This method is an example of "brute-force" computing. You'll learn in more advanced computer science courses that for many interesting problems there's no known algorithmic approach other than using sheer brute force.

**4.22** ***(Modified Triangle Printing Program)*** Modify Exercise 4.15 to combine your code from the four separate triangles of asterisks such that all four patterns print side by side. [*Hint:* Make clever use of nested `for` loops.]

**4.23** ***(De Morgan's Laws)*** In this chapter, we discussed the logical operators `&&`, `&`, `||`, `|`, `^` and `!`. De Morgan's laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(`*condition1* `&&` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `||` `!`*condition2*`)`. Also, the expression `!(`*condition1* `||` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `&&` `!`*condition2*`)`. Use De Morgan's laws to write equivalent expressions for each of the following, then write an application to show that both the original expression and the new expression in each case produce the same value:

a) `!(x < 5) && !(y >= 7)`
b) `!(a == b) || !(g != 5)`
c) `!((x <= 8) && (y > 4))`
d) `!((i > 4) || (j <= 6))`

**4.24** ***(Diamond Printing Program)*** Write an application that prints the following diamond shape. You may use output statements that print a single asterisk (`*`), a single space or a single newline character. Maximize your use of repetition (with nested `for` statements), and minimize the number of output statements.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

**4.25** ***(Modified Diamond Printing Program)*** Modify the application you wrote in Exercise 4.24 to read an odd number in the range 1–19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

**4.26** A criticism of the `break` statement and the `continue` statement is that each is unstructured. Actually, these statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you'd remove any `break` statement from a loop in a program and replace it with some structured equivalent. [*Hint:* The `break` statement exits a loop from the body of the loop. The other way to exit 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 develop here to remove the `break` statement from the application in Fig. 4.11.

**4.27** 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();
   } // end inner for

   System.out.println();
} // end outer for
```

**4.28** Describe in general how you'd remove any `continue` statement from a loop in a program and replace it with some structured equivalent. Use the technique you develop here to remove the `continue` statement from the program in Fig. 4.12.

**4.29** *("The Twelve Days of Christmas" Song)* Write an application that uses repetition and `switch` statements to print the song "The Twelve Days of Christmas." One `switch` statement should be used to print the day ("first," "second," and so on). A separate `switch` statement should be used to print the remainder of each verse. Visit the website `en.wikipedia.org/wiki/The_Twelve_Days_of_Christmas_(song)` for the lyrics of the song.

## Making a Difference

**4.30** *(Global Warming Facts Quiz)* The controversial issue of global warming has been widely publicized by the film "An Inconvenient Truth," featuring former Vice President Al Gore. Mr. Gore and a U.N. network of scientists, the Intergovernmental Panel on Climate Change, shared the 2007 Nobel Peace Prize in recognition of "their efforts to build up and disseminate greater knowledge about man-made climate change." Research *both* sides of the global warming issue online (you might want to search for phrases like "global warming skeptics"). Create a five-question multiple-choice quiz on global warming, each question having four possible answers (numbered 1–4). Be objective and try to fairly represent both sides of the issue. Next, write an application that administers the quiz, calculates the number of correct answers (zero through five) and returns a message to the user. If the user correctly answers five questions, print "Excellent"; if four, print "Very good"; if three or fewer, print "Time to brush up on your knowledge of global warming," and include a list of some of the websites where you found your facts.

**4.31** *(Tax Plan Alternatives; The "FairTax")* There are many proposals to make taxation fairer. Check out the FairTax initiative in the United States at `www.fairtax.org`. Research how the proposed FairTax works. One suggestion is to eliminate income taxes and most other taxes in favor of a 23% consumption tax on all products and services that you buy. Some FairTax opponents question the 23% figure and say that because of the way the tax is calculated, it would be more accurate to say the rate is 30%—check this carefully. Write a program that prompts the user to enter expenses in various expense categories they have (e.g., housing, food, clothing, transportation, education, health care, vacations), then prints the estimated FairTax that person would pay.

**4.32** *(Facebook User Base Growth)* According to CNNMoney.com, Facebook hit one billion users in October 2012. Using the compound-growth technique you learned in Fig. 4.6 and assuming its user base grows at a rate of 4% per month, how many months will it take for Facebook to grow its user base to 1.5 billion users? How many months will it take for Facebook to grow its user base to two billion users?