Control Statements: Part 2

OBJECTIVES

In this Chapter you'll learn:

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

5.1 Introduction

- `for` repetition statement
- `do`…`while` repetition statement
- `switch` multiple-selection statement
- `break` statement
- `continue` statement
- Logical operators
- Control statements summary.

5.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - a **control variable** (or loop counter)
  - the **initial value** of the control variable
  - the **increment** (or **decrement**) by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
  - the **loop-continuation condition** that determines if looping should continue.

```java
// Fig. 5.1: WhileCounter.java
// Counter-controlled repetition with the while repetition statement.

public class WhileCounter
{
   public static void main( String[] args )
   {
      int counter = 1; // declare and initialize control variable

      while ( counter <= 10 ) // loop-continuation condition
      {
         System.out.printf( "%d  ", counter );
         ++counter; // increment control variable by 1
      } // end while

      System.out.println(); // output a newline
   } // end main
} // end class WhileCounter
```

Declares and initializes control variable `counter` to 1

Loop-continuation condition tests for `count`'s final value

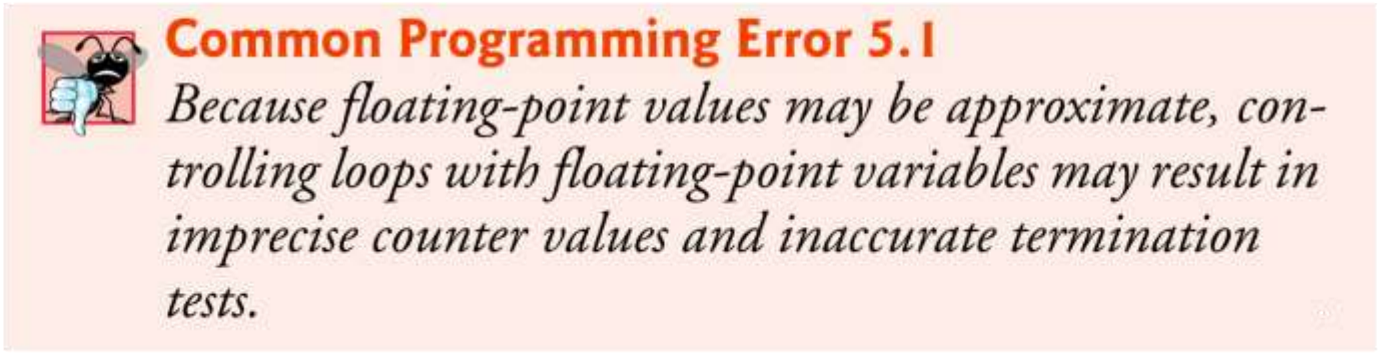
Initializes `gradeCounter` to 1; indicates first grade about to be input

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

Fig. 5.1 | Counter-controlled repetition with the `while` repetition statement.

5.2 Essentials of Counter-Controlled Repetition (Cont.)

- In Fig. 5.1, 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`.
- 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`).
- Line 13 increments the control variable by 1 for each iteration of the loop.



Common Programming Error 5.1

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 5.1

Use integers to control counting loops.

Software Engineering Observation 5.1

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

5.3 for Repetition Statement

- `for` **repetition statement**
  - Specifies the counter-controlled-repetition details in a single line of code.
  - Figure 5.2 reimplements the application of Fig. 5.1 using `for`.

```java
// Fig. 5.2: ForCounter.java
// Counter-controlled repetition with the for repetition statement.

public class ForCounter
{
   public static void main( String[] args )
   {
      // for statement header includes initialization,
      // loop-continuation condition and increment
      for ( int counter = 1; counter <= 10; counter++ )
         System.out.printf( "%d  ", counter );

      System.out.println(); // output a newline
   } // end main
} // end class ForCounter
```

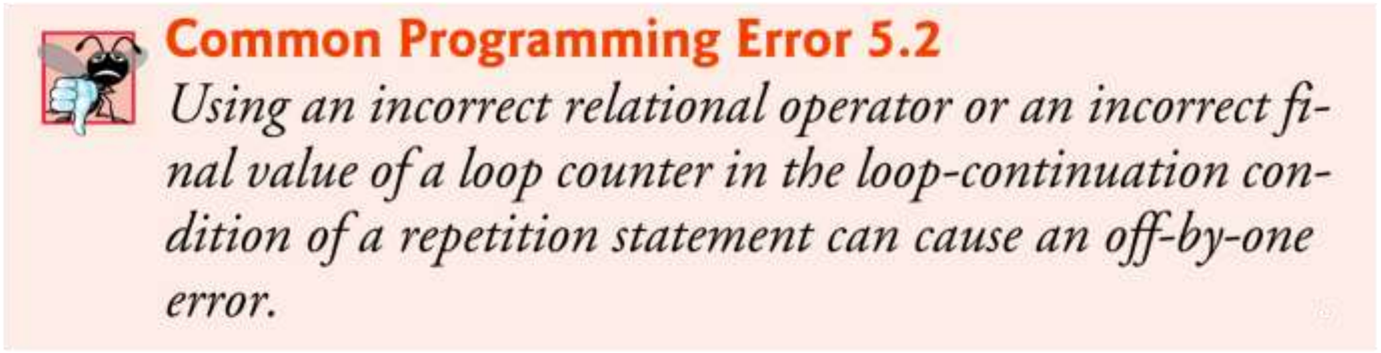
for statement's header contains everything you need for counter-controlled repetition

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

Fig. 5.2 | Counter-controlled repetition with the `for` repetition statement.

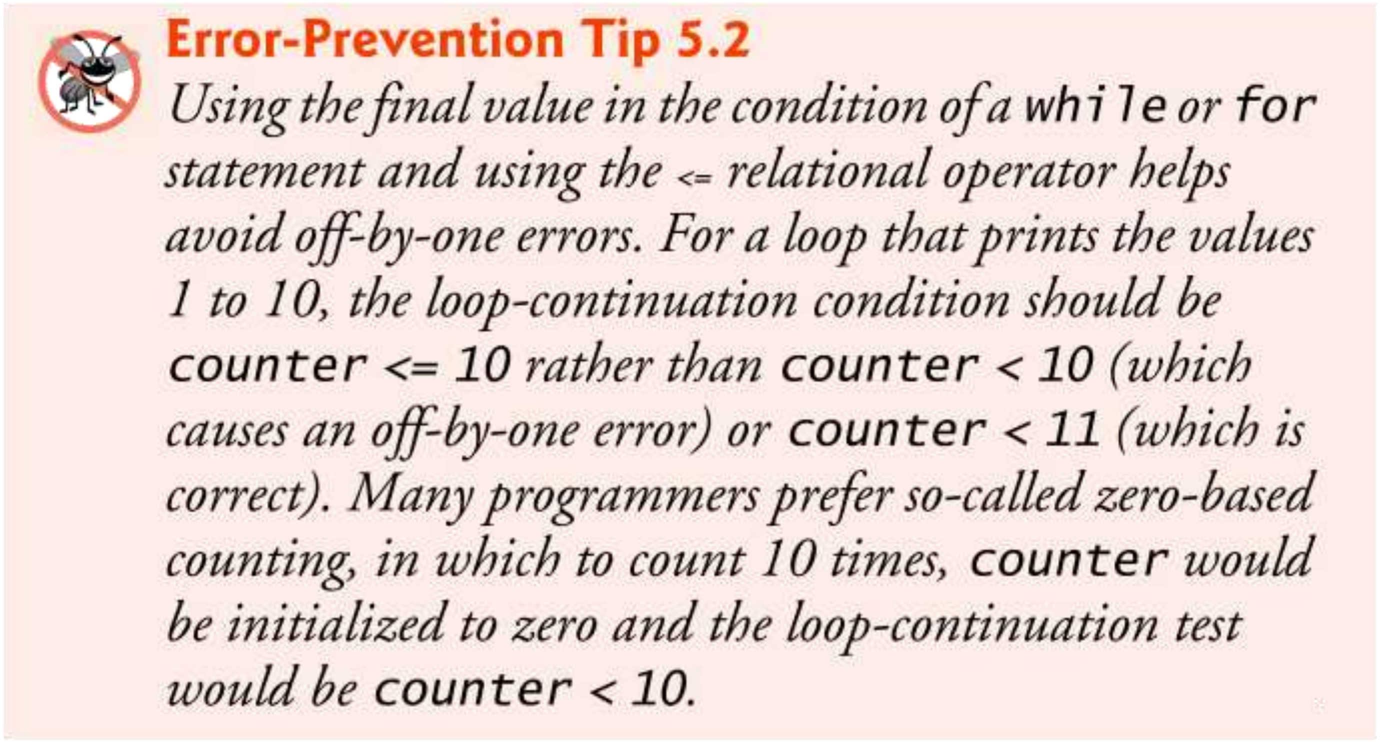
5.3 `for` Repetition Statement (Cont.)

- When the `for` statement begins executing, the control variable is declared and initialized.
- Next, the program checks the loop-continuation condition, which is between the two required semicolons.
- If the condition initially is true, the body statement executes.
- After executing the loop's body, the program increments the control variable in the increment expression, 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.
- A common logic error with counter-controlled repetition is an **off-by-one error**.



Common Programming Error 5.2

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 5.2

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

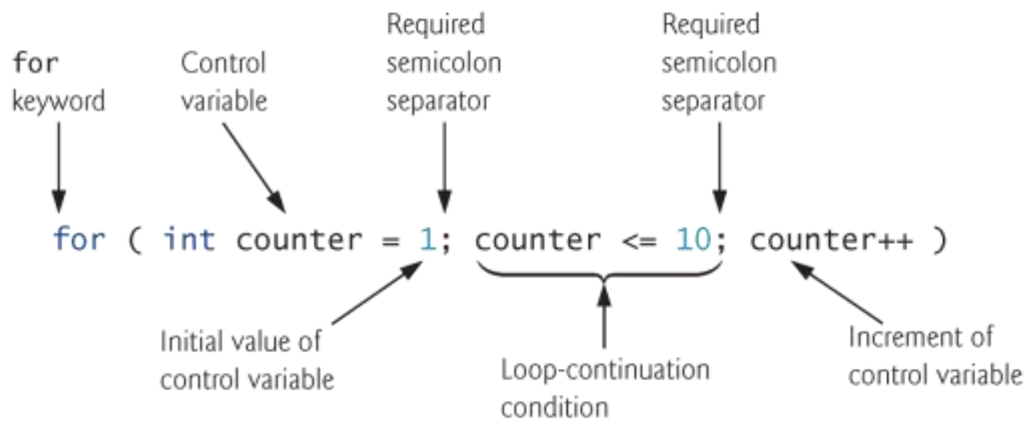

for keyword
Control variable
Required semicolon separator
Required semicolon separator
for (int counter = 1; counter <= 10; counter++)
Initial value of control variable
Loop-continuation condition
Increment of control variable

Fig. 5.3 | for statement header components.

5.3 `for` Repetition Statement (Cont.)

- The general format of the `for` statement is

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

  - the *initialization* expression names the loop's control variable and optionally provides its initial value
  - *loopContinuationCondition* determines whether the loop should continue executing
  - *increment* modifies the control variable's value (possibly an increment or decrement), so that the loop-continuation condition eventually becomes false.
- The two semicolons in the `for` header are required.

5.3 for Repetition Statement (Cont.)

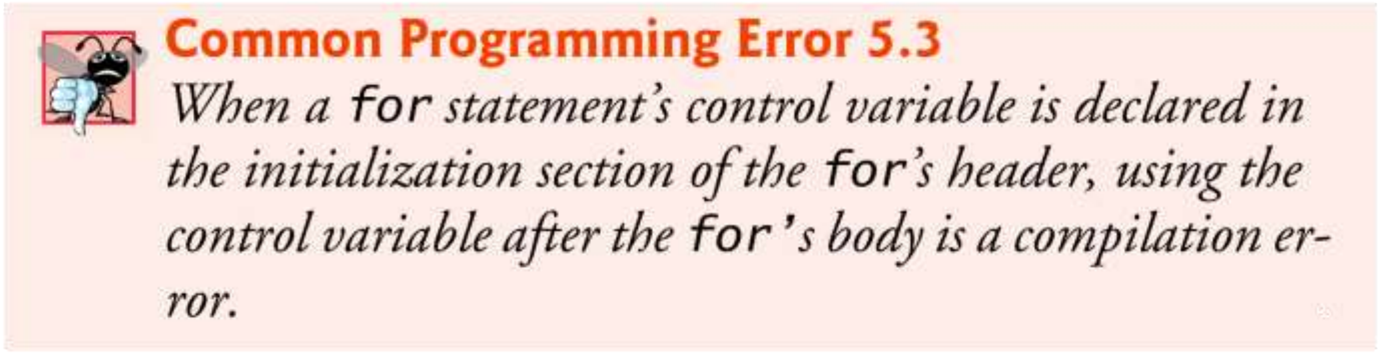
- In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
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.

5.3 for Repetition Statement (Cont.)

- A variable's **scope** defines where it can be used in a program.
  - 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.



Common Programming Error 5.3

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.*

5.3 for Repetition Statement (Cont.)

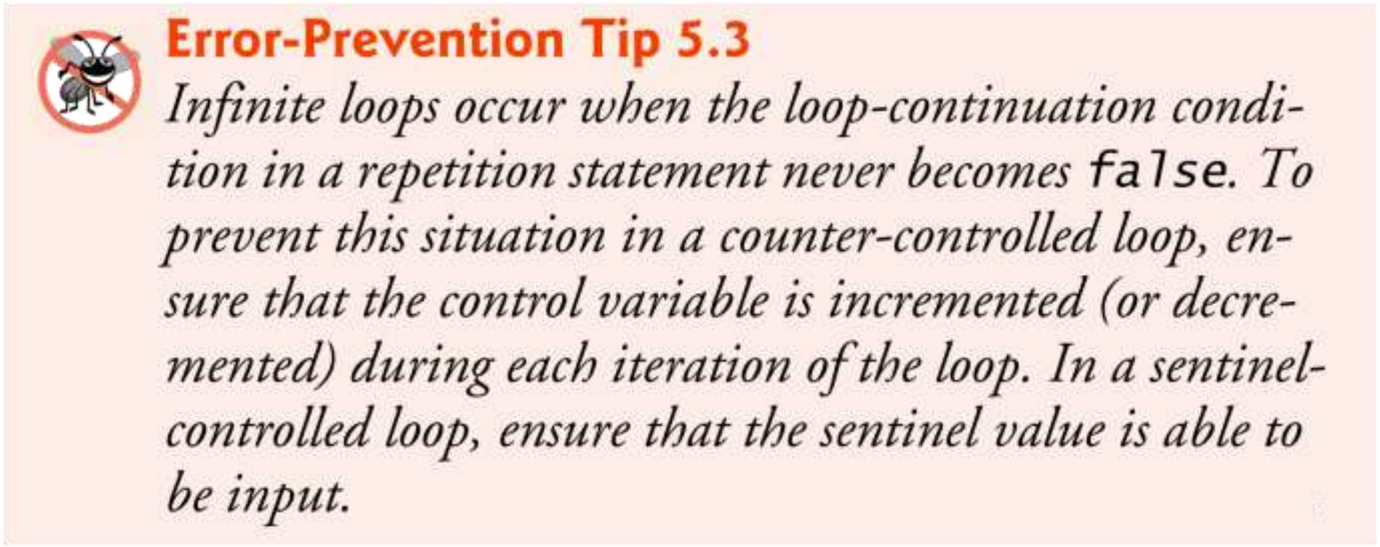
- All three expressions in a `for` header are optional.
  - If the *loopContinuationCondition* is omitted, the condition is always true, thus creating an infinite loop.
  - You might omit the *initialization* expression if the program initializes the control variable before the loop.
  - You might omit the *increment* if the program calculates it 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, so

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

are equivalent increment expressions in a `for` statement.

Common Programming Error 5.4

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 5.3

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 incremented (or decremented) during each iteration of the loop. In a sentinel-controlled loop, ensure that the sentinel value is able to be input.*

5.3 for Repetition Statement (Cont.)

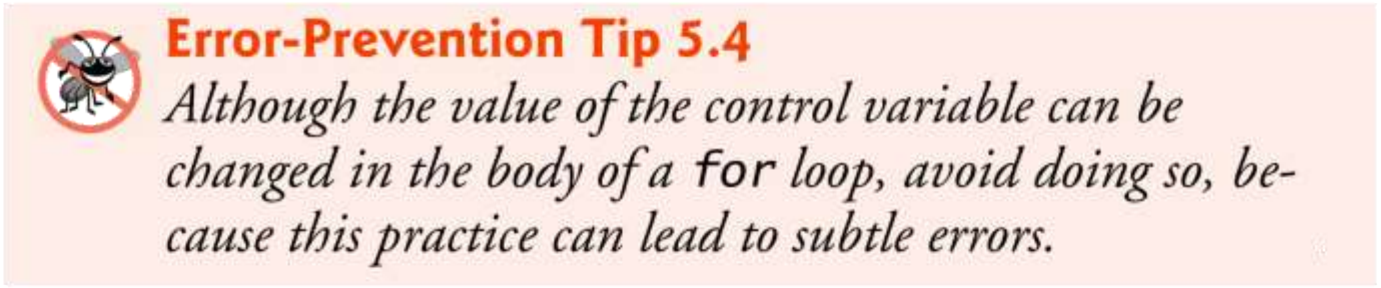
- The initialization, loop-continuation condition and increment can contain arithmetic expressions.
- For example, assume that x = 2 and y = 10. If x and y are not modified in the body of the loop, the statement

```
    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 be negative, in which case it's a decrement, and the loop counts downward.



Error-Prevention Tip 5.4

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.*

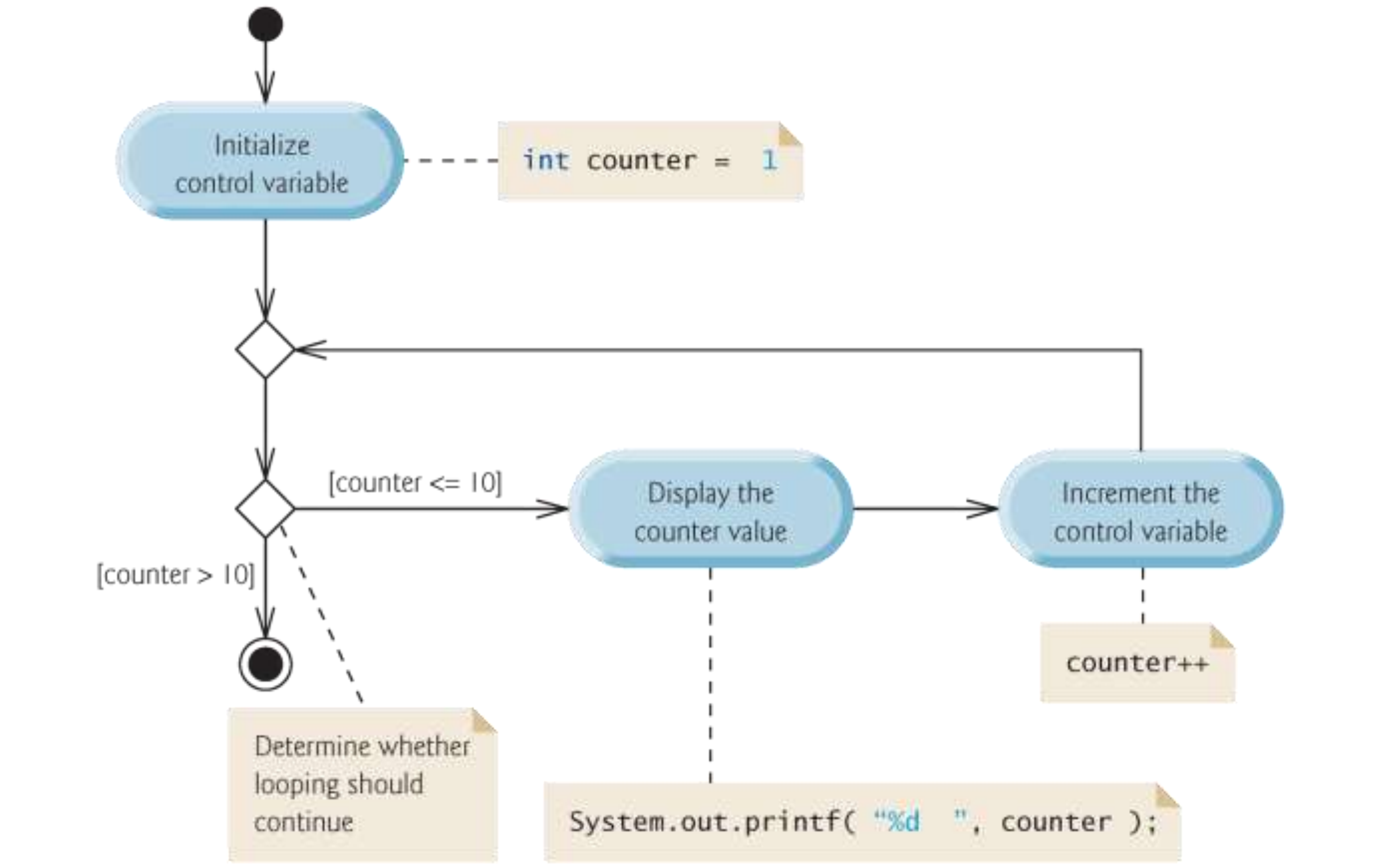

Initialize control variable
int counter = 1
[counter <= 10]
Display the counter value
Increment the control variable
[counter > 10]
counter++
Determine whether looping should continue
System.out.printf("%d ", counter);

Fig. 5.4 | UML activity diagram for the **for** statement in Fig. 5.2.

5.4 Examples Using the for Statement

- 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 )
```

5.4 Examples Using the for Statement (Cont.)

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

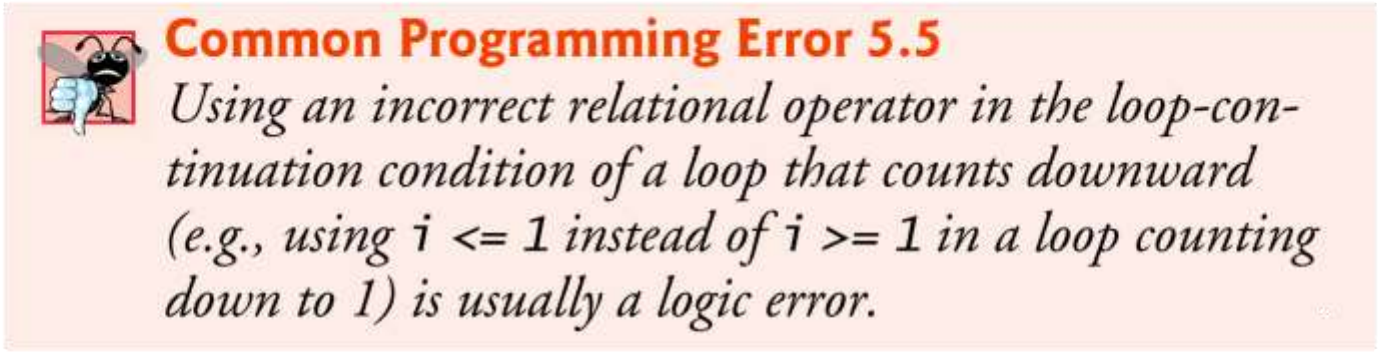
```
for ( int i = 20; i >= 2; i -= 2 )
```

- 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 5.5

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.*

```java
// Fig. 5.5: Sum.java
// Summing integers with the for statement.

public class Sum
{
   public static void main( String[] args )
   {
      int total = 0; // initialize total

      // total even integers from 2 through 20
      for ( int number = 2; number <= 20; number += 2 )
         total += number;

      System.out.printf( "Sum is %d\n", total ); // display results
   } // end main
} // end class Sum
```

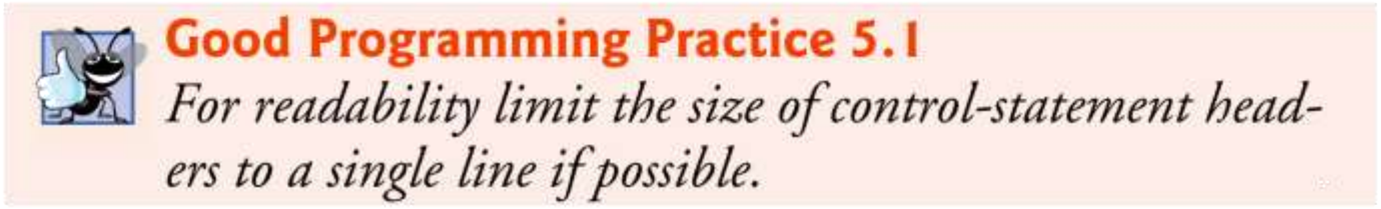
```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

5.4 Examples Using the for Statement (Cont.)

- The *initialization* and *increment* expressions can be comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions.
- Although this is discouraged, the body of the `for` statement in lines 11–12 of Fig. 5.5 could be merged 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 5.1

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

5.4 Examples Using the for Statement (Cont.)

- Compound interest application
- *A person invests $1000 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 nth year.

5.4 Examples Using the `for` Statement (Cont.)

- The solution to this problem (Fig. 5.6) involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.
- Java treats floating-point constants like `1000.0` and `0.05` as type `double`.
- Java treats whole-number constants like `7` and `-22` as type `int`.

```
// Fig. 5.6: Interest.java
// Compound-interest calculations with for.

public class Interest
{
   public static void main( String[] args )
   {
      double amount; // amount on deposit at end of each year
      double principal = 1000.0; // initial amount before interest
      double rate = 0.05; // interest rate

      // display headers
      System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );

      // calculate amount on deposit for each of ten years
      for ( int year = 1; year <= 10; year++ )
      {
         // calculate new amount for specified year
         amount = principal * Math.pow( 1.0 + rate, year );

```

Java treats floating-point literals as `double` values

Uses `static` method `Math.pow` to help calculate the amount on deposit

Fig. 5.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

```java
            // display the year and the amount
            System.out.printf( "%4d%,20.2f\n", year, amount );
         } // end for
      } // end main
   } // end class Interest
```

Comma in format specifier indicates that large numbers should be displayed with thousands separators

```
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
```

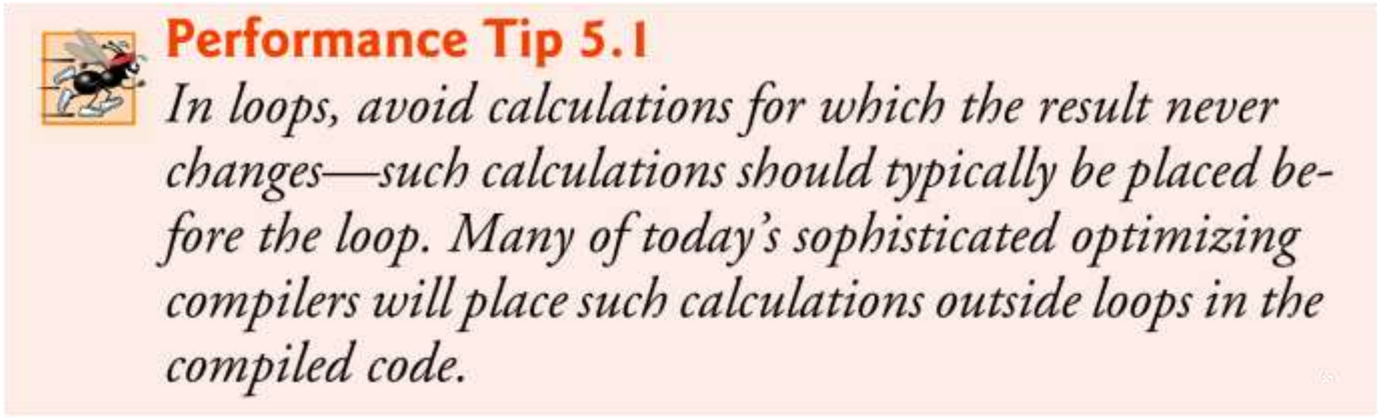
Fig. 5.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

5.4 Examples Using the for Statement (Cont.)

- In the format specifier `%20s`, the integer `20` between the `%` and the conversion character `s` indicates that the value output 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, the value is **right justified** in the field by default.
- If the `year` value to be output were more thanhas more characters than the field width, the field width would be extended to the right to accommodate the entire value.
- To indicate that values should be output **left justified**, precede the field width with the **minus sign (–) formatting flag** (e.g., `%-20s`).

5.4 Examples Using the for Statement (Cont.)

- Classes provide methods that perform common tasks on objects.
- Most methods must be called on a specific object.
- Many classes also provide methods that perform common tasks and do not require objects. These are called `static` methods.
- Java does not include an exponentiation operator—`Math` class `static` method `pow` can be used 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* `)`
- `Math.pow(`*x*`,` *y*`)` calculates the value of x raised to the yth power. The method receives two `double` arguments and returns a `double` value.

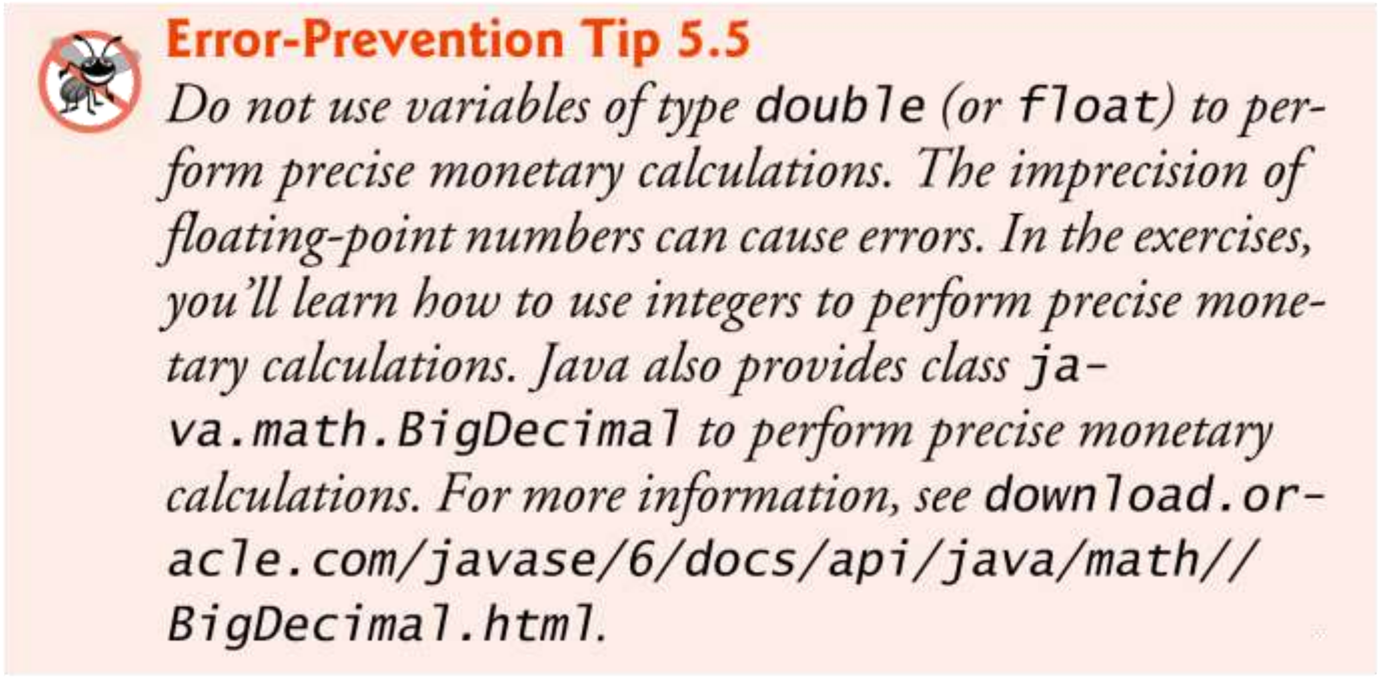


Performance Tip 5.1

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.

5.4 Examples Using the `for` Statement (Cont.)

- In the format specifier `%,20.2f`, the **comma (,) formatting flag** indicates that the floating-point value should be output with a **grouping separator.**
- Separator is specific to the user's locale (i.e., country).
- 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.



Error-Prevention Tip 5.5

Do not use variables of type `double` *(or* `float`*) to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors. In the exercises, you'll learn how to use integers to perform precise monetary calculations. Java also provides class* `java.math.BigDecimal` *to perform precise monetary calculations. For more information, see* `download.oracle.com/javase/6/docs/api/java/math//BigDecimal.html`.

5.5 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.

```java
// Fig. 5.7: DoWhileTest.java
// do...while repetition statement.

public class DoWhileTest
{
   public static void main( String[] args )
   {
      int counter = 1; // initialize counter

      do
      {
         System.out.printf( "%d  ", counter );
         ++counter;
      } while ( counter <= 10 ); // end do...while

      System.out.println(); // outputs a newline
   } // end main
} // end class DoWhileTest
```

Condition tested at end of loop, so loop always executes at least once

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

Fig. 5.7 | do...while repetition statement.

5.5 do…while Repetition Statement (Cont.)

- Figure 5.8 contains the UML activity diagram for the `do`…`while` statement.
- The diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once.

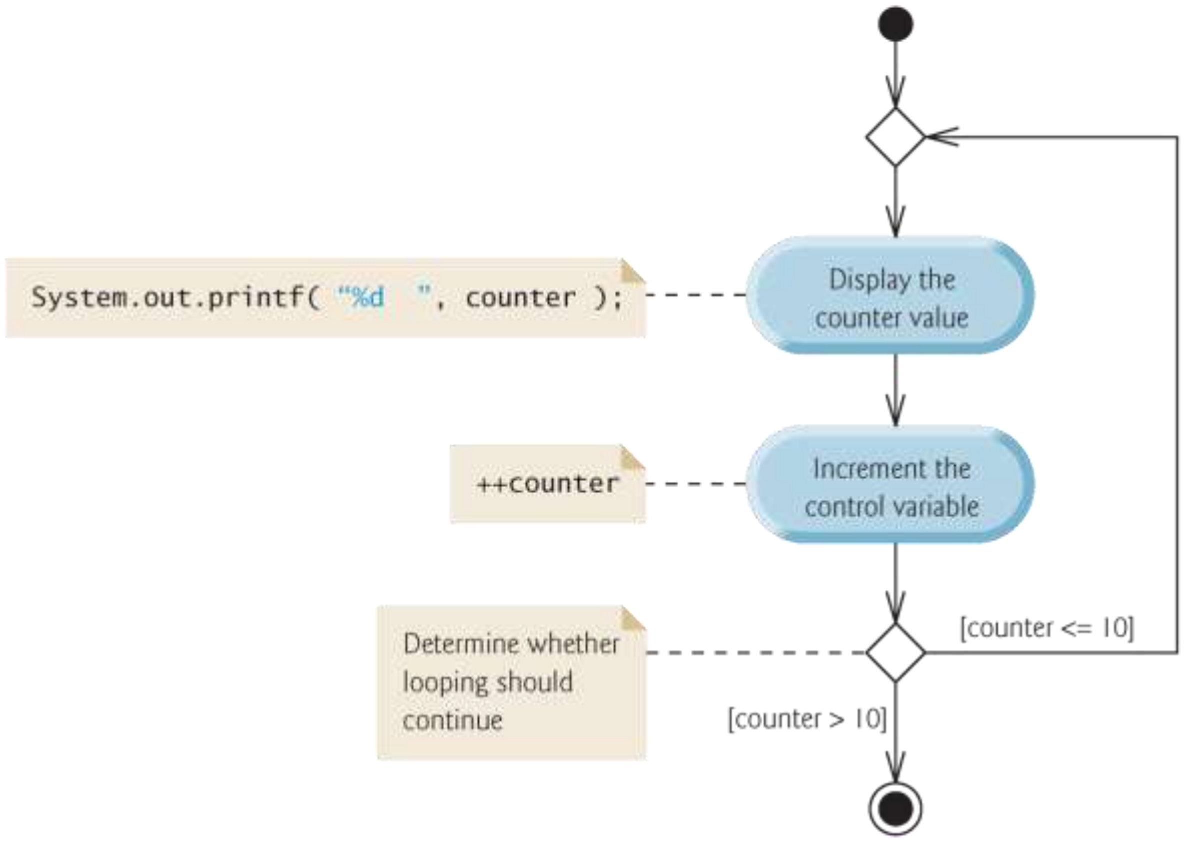
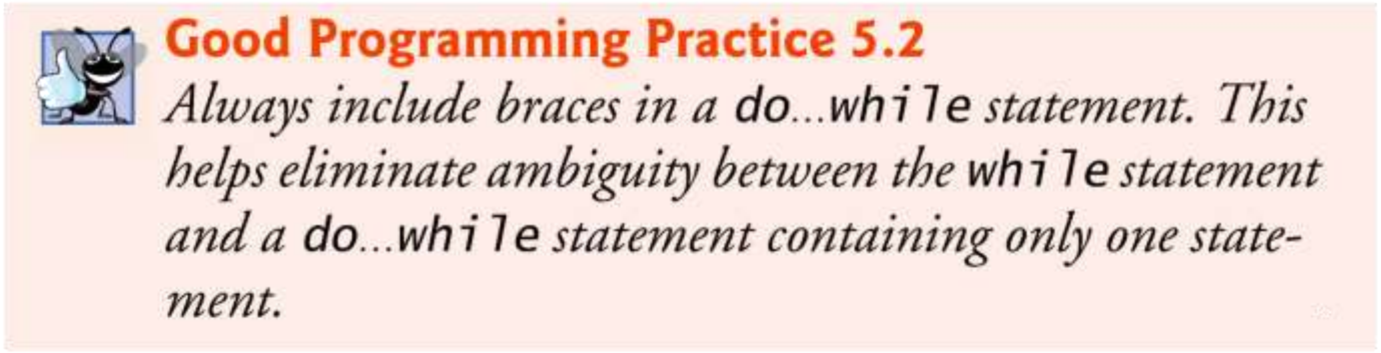

System.out.printf("%d ", counter);
Display the counter value
++counter
Increment the control variable
Determine whether looping should continue
[counter <= 10]
[counter > 10]

Fig. 5.8 | do...while repetition statement UML activity diagram.

5.5 do…while Repetition Statement (Cont.)

- Braces are not required in the `do…while` repetition statement if there’s only one statement in the body.
- Most programmers include the braces, to avoid confusion between the `while` and `do…while` statements.
- Thus, the `do…while` statement with one body statement is usually written as follows:

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



Good Programming Practice 5.2

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.*

5.6 switch Multiple-Selection Statement

- switch **multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char`.

```java
// Fig. 5.9: GradeBook.java
// GradeBook class uses switch statement to count letter grades.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents
   // int instance variables are initialized to 0 by default
   private int total; // sum of grades
   private int gradeCounter; // number of grades entered
   private int aCount; // count of A grades
   private int bCount; // count of B grades
   private int cCount; // count of C grades
   private int dCount; // count of D grades
   private int fCount; // count of F grades

   // constructor initializes courseName;
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 1 of 7.)

```
   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 2 of 7.)

```
      // input arbitrary number of grades from user
      public void inputGrades()
      {
         Scanner input = new Scanner( System.in );

         int grade; // grade entered by user

         System.out.printf( "%s\n%s\n   %s\n   %s\n",
            "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" );

```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 3 of 7.)

```
      // loop until user enters the end-of-file indicator
      while ( input.hasNext() )
      {
         grade = input.nextInt(); // read grade
         total += grade; // add grade to total
         ++gradeCounter; // increment number of grades

         // call method to increment appropriate counter
         incrementLetterGradeCounter( grade );
      } // end while
   } // end method inputGrades

```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 4 of 7.)

```
   // add 1 to appropriate counter for specified grade
   private void incrementLetterGradeCounter( int grade )
   {
      // determine which grade was entered
      switch ( grade / 10 )
      {
         case 9:  // grade was between 90
         case 10: // and 100, inclusive
            ++aCount; // increment aCount
            break; // necessary to exit switch

         case 8: // grade was between 80 and 89
            ++bCount; // increment bCount
            break; // exit switch

         case 7: // grade was between 70 and 79
            ++cCount; // increment cCount
            break; // exit switch

         case 6: // grade was between 60 and 69
            ++dCount; // increment dCount
            break; // exit switch
```

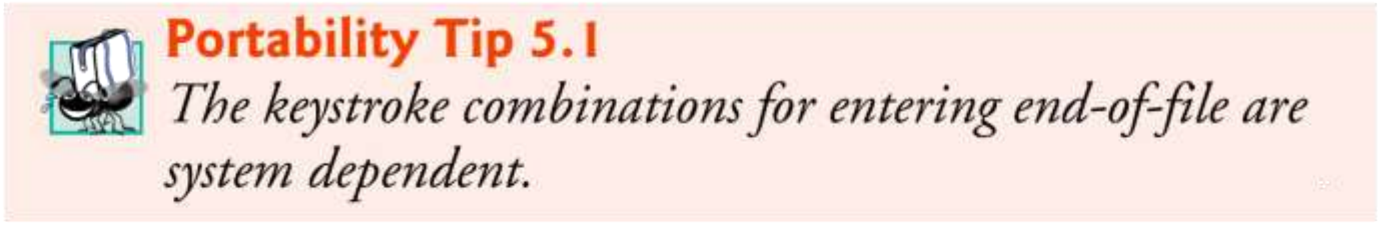
Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 5 of 7.)

```java

         default: // grade was less than 60
            ++fCount; // increment fCount
            break; // optional; will exit switch anyway
      } // end switch
   } // end method incrementLetterGradeCounter

   // display a report based on the grades entered by the user
   public void displayGradeReport()
   {
      System.out.println( "\nGrade Report:" );

      // if user entered at least one grade...
      if ( gradeCounter != 0 )
      {
         // calculate average of all grades entered
         double average = (double) total / gradeCounter;

         // output summary of results
         System.out.printf( "Total of the %d grades entered is %d\n",
            gradeCounter, total );
         System.out.printf( "Class average is %.2f\n", average );
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 6 of 7.)

```
                  "Number of students who received each grade:",
                  "A: ", aCount,   // display number of A grades
                  "B: ", bCount,   // display number of B grades
                  "C: ", cCount,   // display number of C grades
                  "D: ", dCount,   // display number of D grades
                  "F: ", fCount ); // display number of F grades
            } // end if
            else // no grades were entered, so output appropriate message
               System.out.println( "No grades were entered" );
         } // end method displayGradeReport
      } // end class GradeBook
```

Fig. 5.9 | GradeBook class uses switch statement to count letter grades. (Part 7 of 7.)



Portability Tip 5.1

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

Common Programming Error 5.6

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

```java
// Fig. 5.10: GradeBookTest.java
// Create GradeBook object, input grades and display grade report.

public class GradeBookTest
{
   public static void main( String[] args )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.inputGrades(); // read grades from user
      myGradeBook.displayGradeReport(); // display report based on grades
   } // end main
} // end class GradeBookTest
```

Fig. 5.10 | Create **GradeBook** object, input grades and display grade report. (Part 1 of 3.)

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

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
```

Fig. 5.10 | Create **GradeBook** object, input grades and display grade report. (Part 2 of 3.)

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

Fig. 5.10 | Create GradeBook object, input grades and display grade report. (Part 3 of 3.)

5.6 `switch` Multiple-Selection Statement (Cont.)

- The **end-of-file indicator** is a system-dependent keystroke combination which the user enters to indicate that there is no more data to 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, end-of-file can be entered by typing
  - *<Ctrl> z*
- On some systems, you must press *Enter* after typing the end-of-file key sequence.
- Windows typically displays the characters `^Z` on the screen when the end-of-file indicator is typed.

5.6 `switch` Multiple-Selection Statement (Cont.)

- `Scanner` method hasNext determine whether there is more data to input. This method returns the `boolean` value `true` if there is more data; otherwise, it returns `false`.
- As long as the end-of-file indicator has not been typed, method `hasNext` will return `true`.

5.6 `switch` Multiple-Selection Statement (Cont.)

- The `switch` statement consists of a block that contains a sequence of case **labels** and an optional default **case.**
- The program evaluates the **controlling expression** in the parentheses following keyword `switch`.
- The program compares the controlling expression's value (which must evaluate to an integral value of type `byte`, `char`, `short` or `int`) with each `case` label.
- If a match occurs, the program executes that `case`'s statements.
- The break **statement** causes program control to proceed with the first statement after the `switch`.

5.6 `switch` Multiple-Selection Statement (Cont.)

- `switch` does not provide a mechanism for testing ranges of values—every value must be listed in a separate `case` label.
- Note that each `case` can have multiple statements.
- `switch` differs from other control statements in that it does not require braces around multiple statements in a `case`.
- Without `break`, the statements for a matching case and subsequent cases execute until a `break` or the end of the `switch` is encountered. This is called "falling through."
- If no match occurs between the controlling expression's value and a `case` label, the `default` case executes.
- If no match occurs and there is no `default` case, program control simply continues with the first statement after the `switch`.

Software Engineering Observation 5.2

Recall from Chapter 3 that methods declared with access modifier `private` *can be called only by other methods of the class in which the* `private` *methods are declared. Such methods are commonly referred to as* ***utility methods*** *or* ***helper methods*** *because they're typically used to support the operation of the class's other methods.*

5.6 `switch` Multiple-Selection Statement (Cont.)

- Figure 5.11 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`.
- 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`.

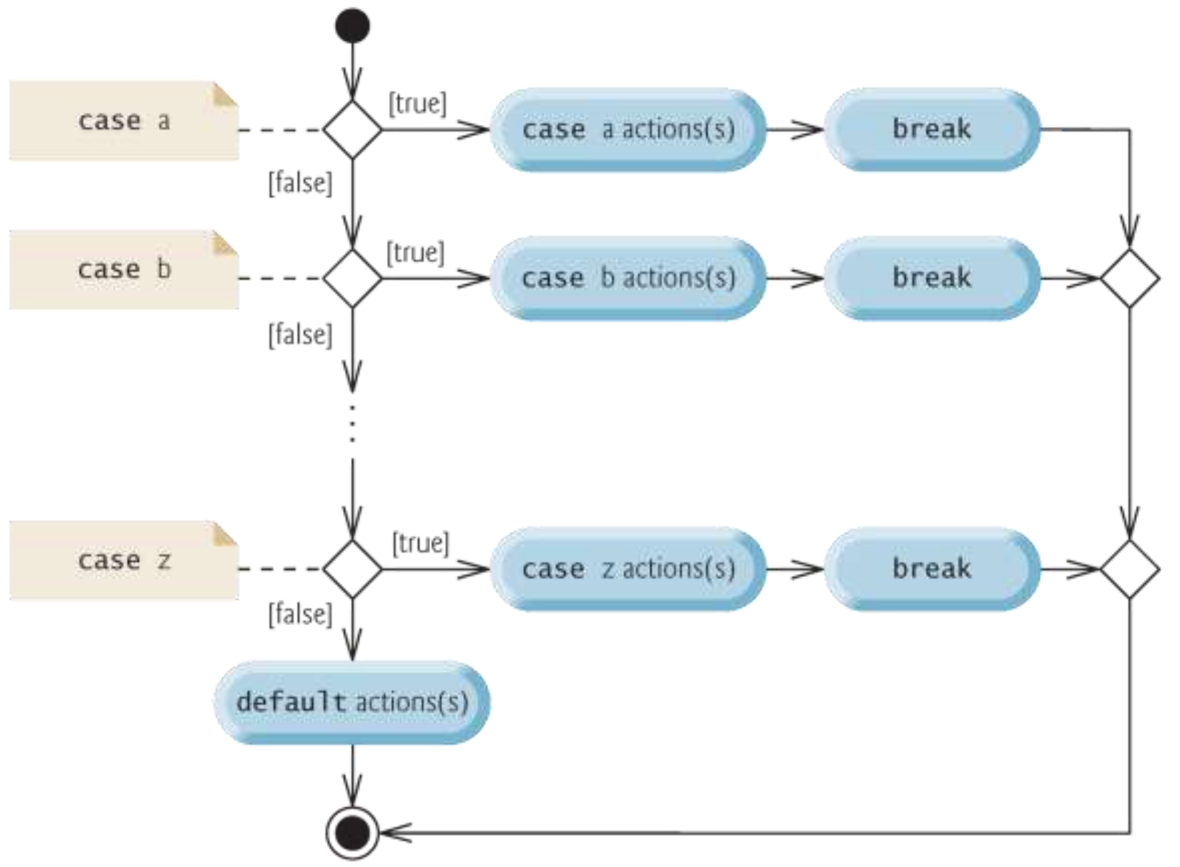
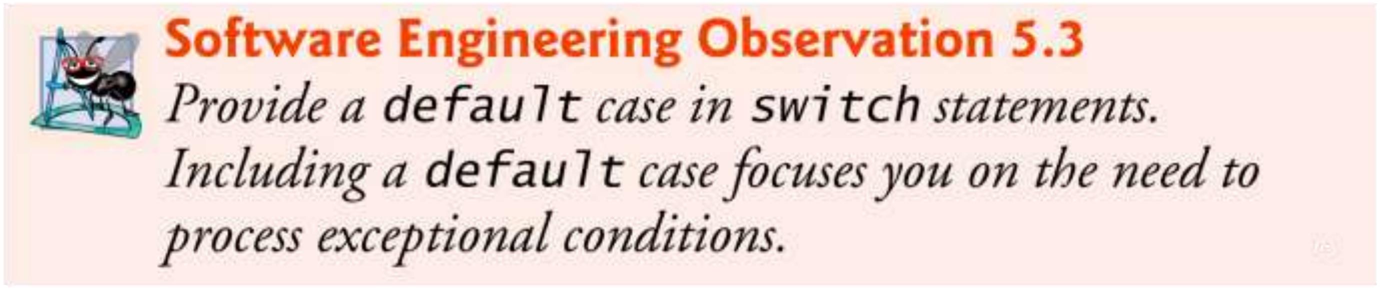
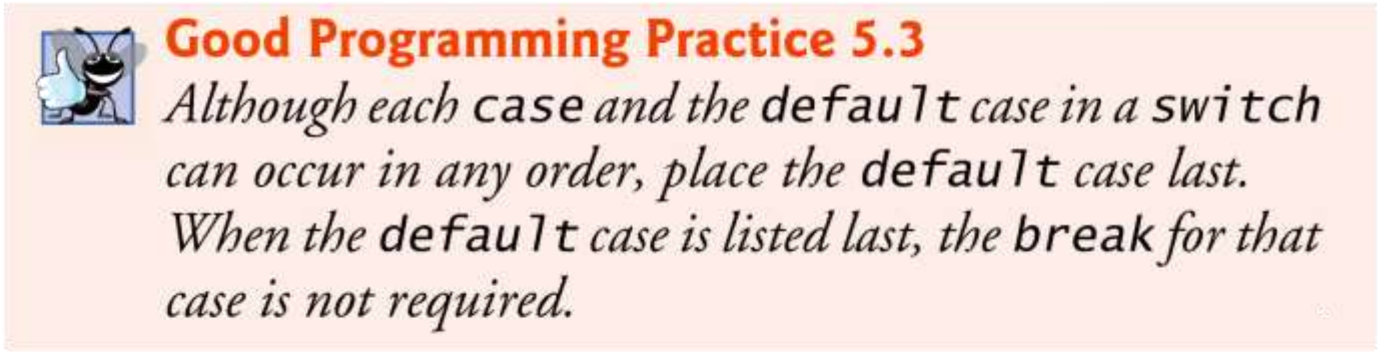


Fig. 5.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.



Software Engineering Observation 5.3

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



Good Programming Practice 5.3

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.*

5.6 `switch` Multiple-Selection Statement (Cont.)

- When using the `switch` statement, remember that each `case` must contain a constant integral expression.
- 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.
- The expression in each `case` can also be a **constant variable**—a variable that contains a value which does not change for the entire program. Such a variable is declared with keyword `final`.
- Java has a feature called enumerations. Enumeration constants can also be used in `case` labels.

5.6 switch Multiple-Selection Statement (Cont.)

- As of Java SE 7, you can use Strings in a switch statement's controlling expression and in case labels as in:
-
```
switch( city )
{
   case "Maynard":
      zipCode = "01754";
      break;
   case "Marlborough":
      zipCode = "01752";
      break;
   case "Framingham":
      zipCode = "01701";
      break;
} // end switch
```
-

5.7 `break` and `continue` Statements

- 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`.

```
1  // Fig. 5.12: 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 ) // if count is 5,
12             break;         // terminate loop
13
14          System.out.printf( "%d ", count );
15       } // end for
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // end main
19 } // end class BreakTest
```

Terminates the loop immediately and program control continues at line 17

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

Fig. 5.12 | break statement exiting a for statement.

5.7 break and continue Statements (Cont.)

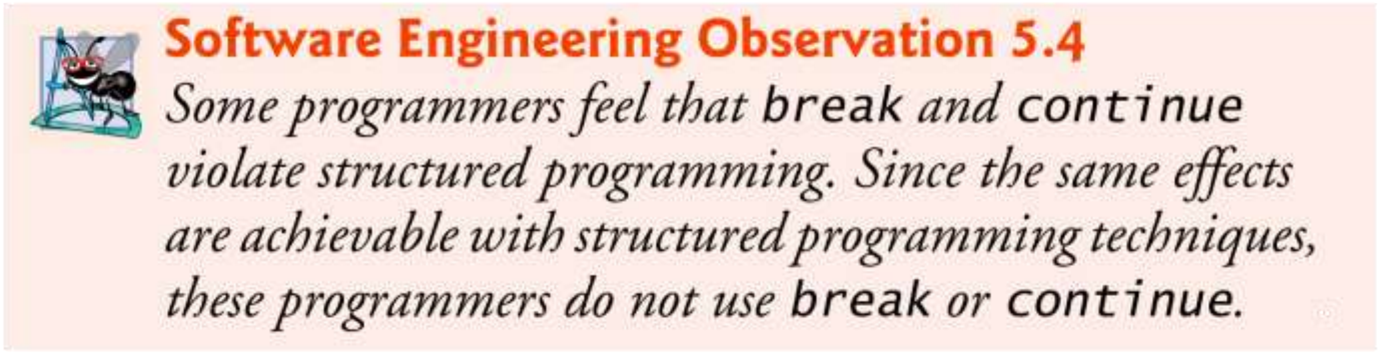
- The `continue` statement, when executed in a `while`, `for` or `do`…`while`, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.
- In `while` and `do`…`while` statements, the program evaluates the loop-continuation test immediately after the `continue`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

```
// Fig. 5.13: ContinueTest.java
// continue statement terminating an iteration of a for statement.
public class ContinueTest
{
   public static void main( String[] args )
   {
      for ( int count = 1; count <= 10; count++ ) // loop 10 times
      {
         if ( count == 5 ) // if count is 5,
            continue; // skip remaining code in loop

         System.out.printf( "%d ", count );
      } // end for

      System.out.println( "\nUsed continue to skip printing 5" );
   } // end main
} // end class ContinueTest
```

Terminates current iteration of loop and proceeds to increment

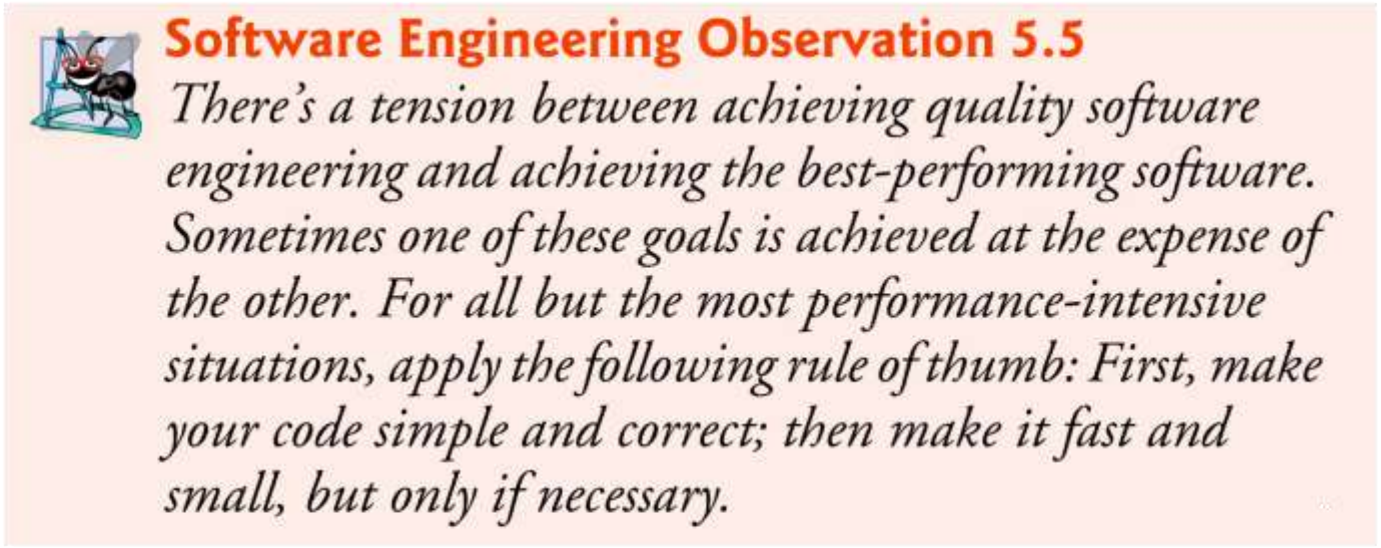
```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.13 | continue statement terminating an iteration of a for statement.



Software Engineering Observation 5.4

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 5.5

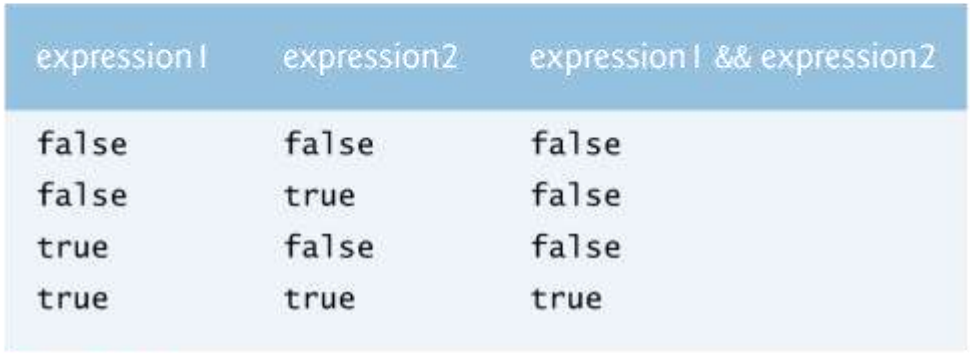
There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and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

5.8 Logical Operators

- 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)
  - ! (logical NOT).
- [*Note:* The &, | and ^ operators are also bitwise operators when they are applied to integral operands.]

5.8 Logical Operators (Cont.)

- The & (**conditional AND**) operator ensures that two conditions are *both true* before choosing a certain path of execution.
- The table in Fig. 5.14 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.

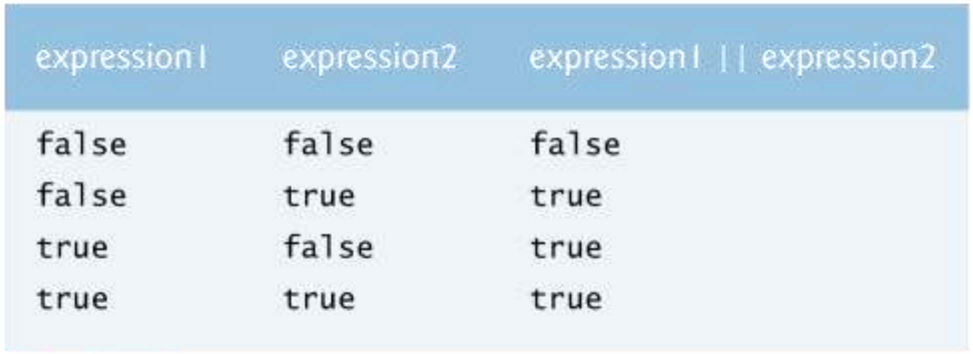


expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 5.14 | && (conditional AND) operator truth table.

5.8 Logical Operators (Cont.)

- The **||** (**conditional OR**) operator ensures that *either or both* of two conditions are true before choosing a certain path of execution.
- Figure 5.15 is a truth table for operator conditional OR (||).
- Operator && has a higher precedence than operator ||.
- Both operators associate from left to right.

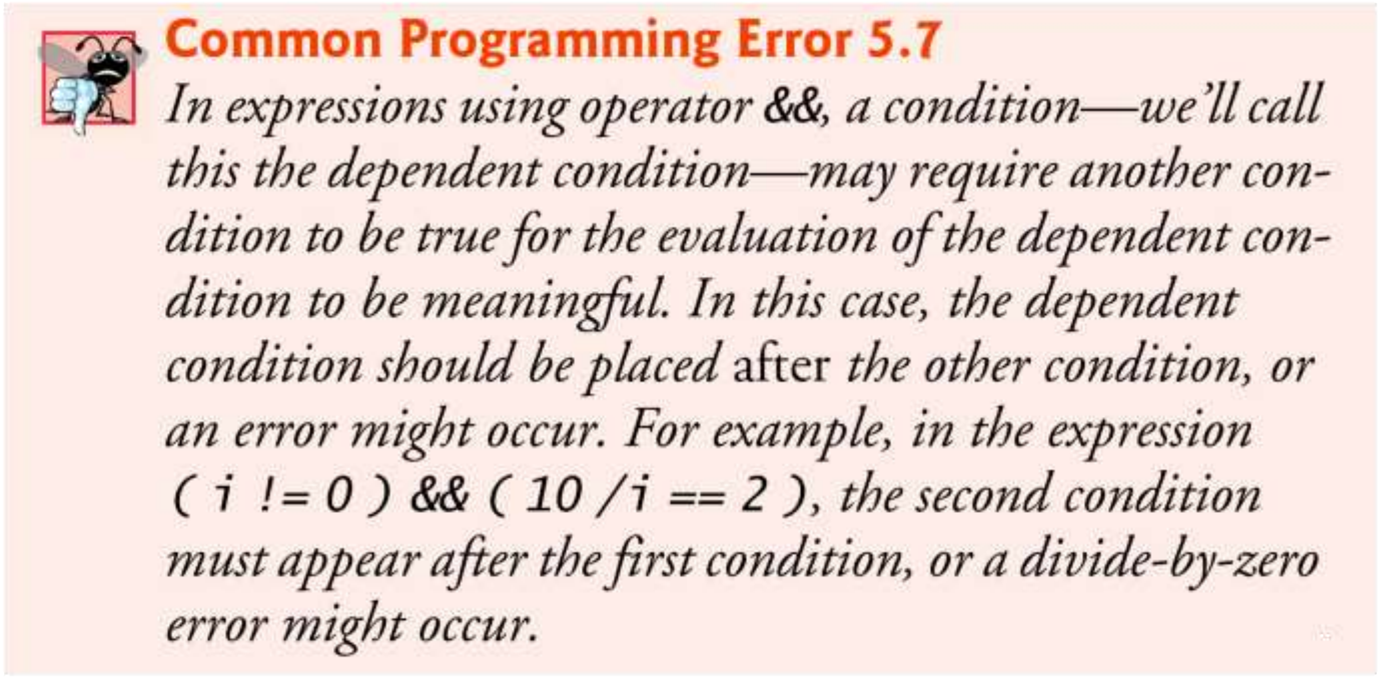


expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 5.15 | || (conditional OR) operator truth table.

5.8 Logical Operators (Cont.)

- The parts of an expression containing && or || operators are evaluated only until it’s known whether the condition is true or false. T
- This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation.**

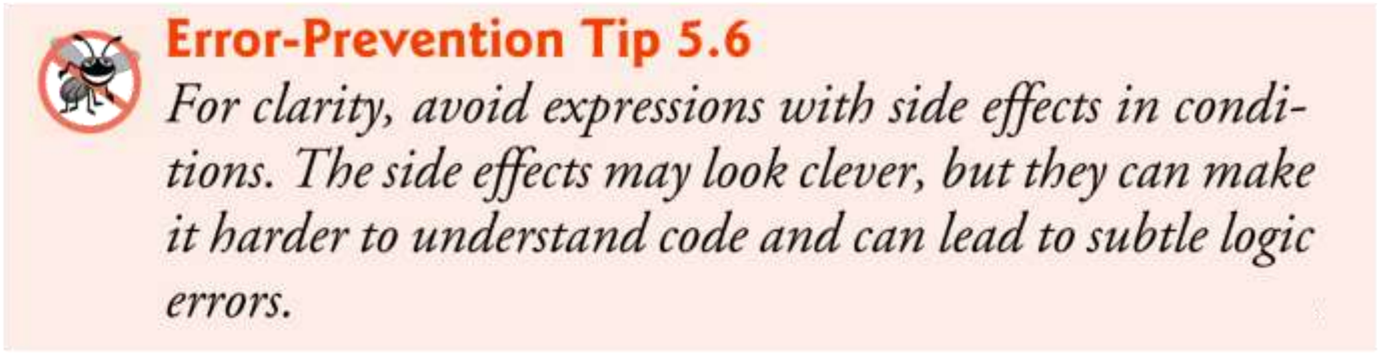


Common Programming Error 5.7

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 other condition, or an error might occur. For example, in the expression* `( i != 0 ) && ( 10 / i == 2 )`*, the second condition must appear after the first condition, or a divide-by-zero error might occur.*

5.8 Logical Operators (Cont.)

- 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).
- This is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator has a required **side effect**—a modification of a variable's value.



Error-Prevention Tip 5.6

For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they can make it harder to understand code and can lead to subtle logic errors.

5.8 Logical Operators (Cont.)

- 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 5.16 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. 5.16 | ^ (boolean logical exclusive OR) operator truth table.

5.8 Logical Operators (Cont.)

- The ! (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition.
- The logical negation operator is a unary operator that has only a single condition as an operand.
- The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`.
- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator.
- Figure 5.17 is a truth table for the logical negation operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. 5.17 | `!` (logical negation, or logical NOT) operator truth table.

5.8 Logical Operators (Cont.)

- Figure 5.18 produces the truth tables discussed in this section.
- The `%b` **format specifier** displays the word "true" or the word "false" based on a `boolean` expression's value.

```java
// Fig. 5.18: LogicalOperators.java
// Logical operators.

public class LogicalOperators
{
   public static void main( String[] args )
   {
      // create truth table for && (conditional AND) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Conditional AND (&&)", "false && false", ( false && false ),
         "false && true", ( false && true ),
         "true && false", ( true && false ),
         "true && true", ( true && true ) );

      // create truth table for || (conditional OR) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Conditional OR (||)", "false || false", ( false || false ),
         "false || true", ( false || true ),
         "true || false", ( true || false ),
         "true || true", ( true || true ) );

```

Value of each condition like this is displayed using format specifier %b

Fig. 5.18 | Logical operators. (Part 1 of 4.)

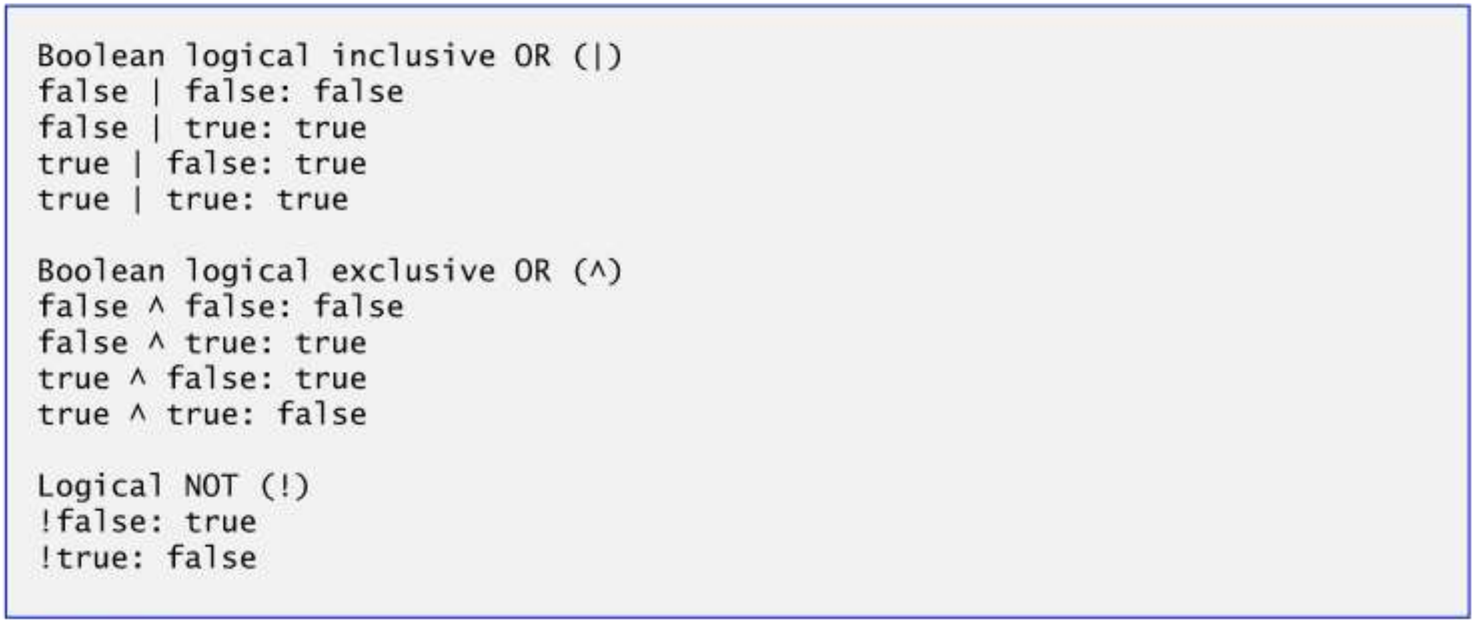
```
      // create truth table for & (boolean logical AND) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Boolean logical AND (&)", "false & false", ( false & false ),
         "false & true", ( false & true ),
         "true & false", ( true & false ),
         "true & true", ( true & true ) );

      // create truth table for | (boolean logical inclusive OR) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Boolean logical inclusive OR (|)",
         "false | false", ( false | false ),
         "false | true", ( false | true ),
         "true | false", ( true | false ),
         "true | true", ( true | true ) );

      // create truth table for ^ (boolean logical exclusive OR) operator
      System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
         "Boolean logical exclusive OR (^)",
         "false ^ false", ( false ^ false ),
         "false ^ true", ( false ^ true ),
         "true ^ false", ( true ^ false ),
         "true ^ true", ( true ^ true ) );

```

Fig. 5.18 | Logical operators. (Part 2 of 4.)

```
        // create truth table for ! (logical negation) operator
        System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
          "!false", ( !false ), "!true", ( !true ) );
     } // end main
  } // 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
```

Fig. 5.18 | Logical operators. (Part 3 of 4.)



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

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

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

Fig. 5.18 | Logical operators. (Part 4 of 4.)

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. 5.19 | Precedence/associativity of the operators discussed so far.

5.9 Structured Programming Summary

- Figure 5.20 uses UML activity diagrams to summarize Java’s control statements.
- Java includes only single-entry/single-exit control statements—there is 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.

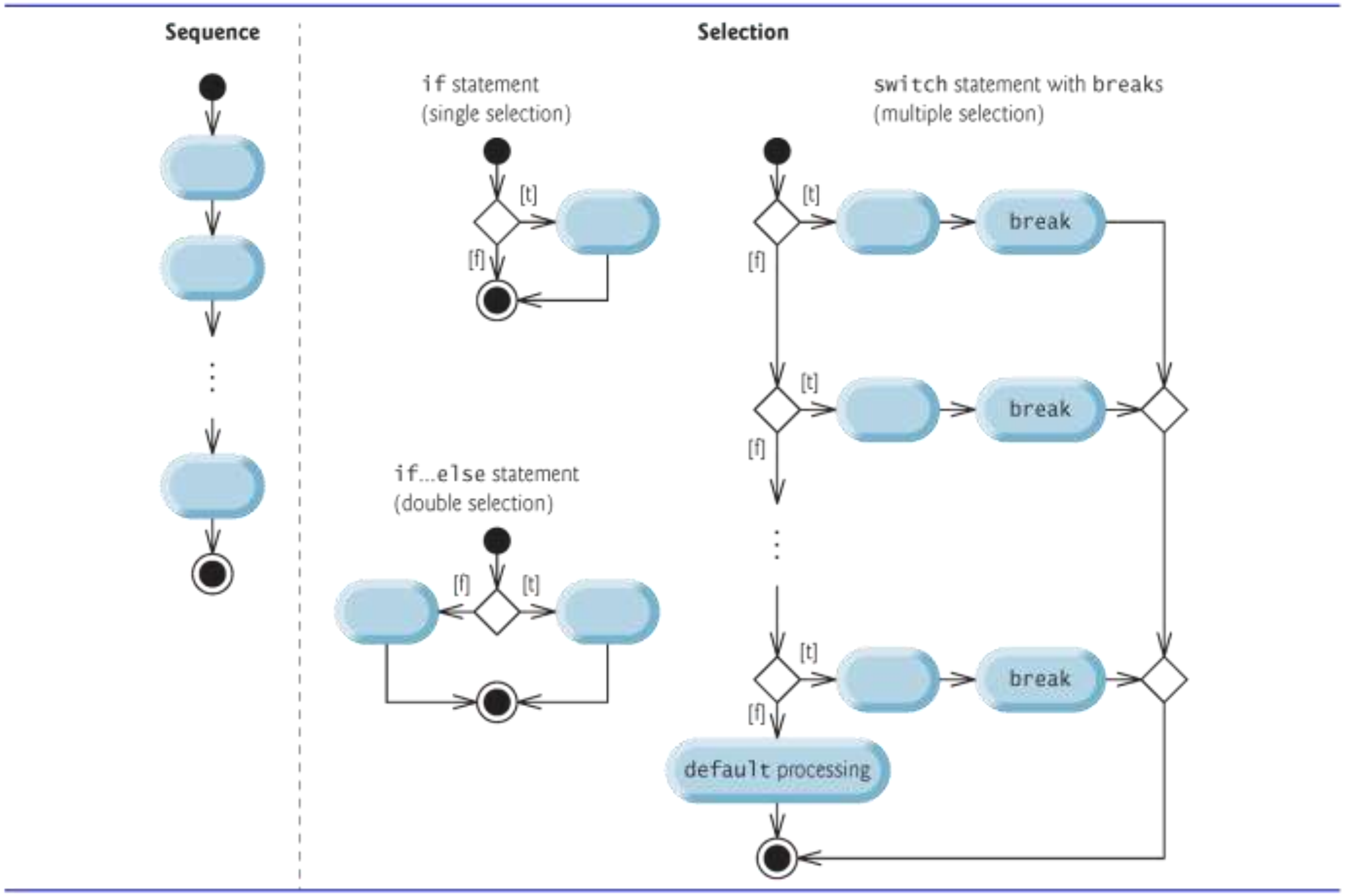


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 1 of 2.)

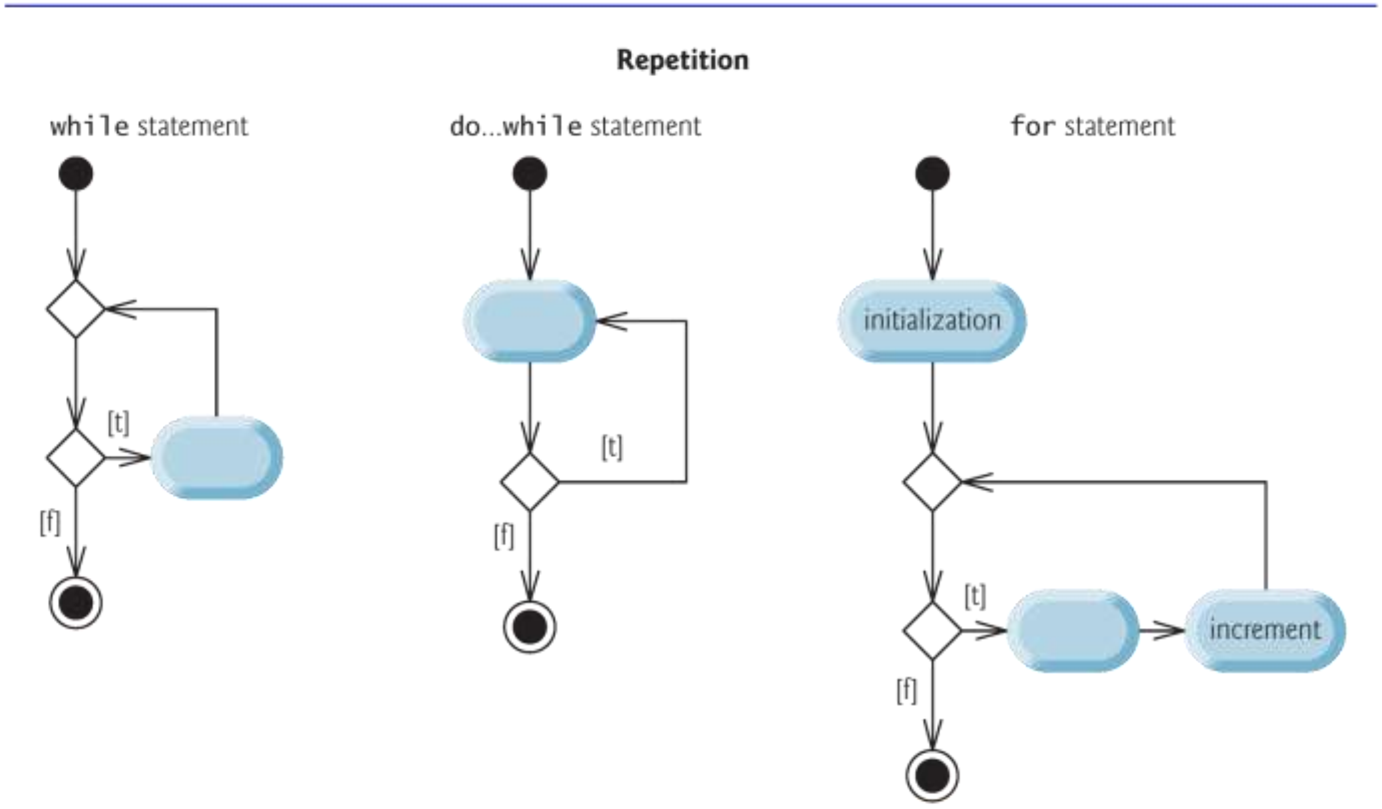


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

5.9 Structured Programming Summary (Cont.)

- Structured programming promotes simplicity.
- Bohm and Jacopini: 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.

5.9 Structured Programming Summary (Cont.)

- Selection is implemented in one of three ways:
  - `if` statement (single selection)
  - `if…else` statement (double selection)
  - `switch` statement (multiple selection)
- 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.

5.9 Structured Programming Summary (Cont.)

- Repetition is implemented in one of three ways:
  - `while` statement
  - `do…while` statement
  - `for` statement
- 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.

5.9 Structured Programming Summary (Cont.)

- 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.