# CHAPTER 3

# SELECTION STATEMENTS

## Objectives

- To declare `boolean` type and write Boolean expressions (§3.2).
- To distinguish between conditional and unconditional `&&` and `||` operators (§3.2.1).
- To use Boolean expressions to control selection statements (§§3.3–3.5).
- To implement selection control using `if` and nested `if` statements (§3.3).
- To implement selection control using `switch` statements (§3.4).
- To write expressions using the conditional operator (§3.5).
- To display formatted output using the `System.out.printf` method and to format strings using the `String.format` method (§3.6).
- To know the rules governing operand evaluation order, operator precedence, and operator associativity (§§3.7–3.8).

## 3.1 Introduction

why selection?
pseudocode

In Chapter 2, "Primitive Data Types and Operations," if you assigned a negative value for `radius` in Listing 2.1, ComputeArea.java, the program would print an invalid result. If the radius is negative, you don't want the program to compute the area. Like all high-level programming languages, Java provides selection statements that let you choose actions with two or more alternative courses. You can use selection statements in the following *pseudocode* (i.e., natural language mixed with programming code) to rewrite Listing 2.1:

```
if the radius is negative
  the program displays a message indicating a wrong input;
else
  the program computes the area and displays the result;
```

Selection statements use conditions. Conditions are Boolean expressions. This chapter first introduces Boolean types, values, operators, and expressions.

## 3.2 `boolean` Data Type and Operations

comparison operators

Often in a program you need to compare two values, such as whether `i` is greater than `j`. Java provides six *comparison operators* (also known as *relational operators*), shown in Table 3.1, which can be used to compare two values. The result of the comparison is a Boolean value: `true` or `false`. For example, the following statement displays `true`:

```
System.out.println(1 < 2);
```

**TABLE 3.1** Comparison Operators

| *Operator* | *Name* | *Example* | *Answer* |
|---|---|---|---|
| `<` | less than | `1 < 2` | `true` |
| `<=` | less than or equal to | `1 <= 2` | `true` |
| `>` | greater than | `1 > 2` | `false` |
| `>=` | greater than or equal to | `1 >= 2` | `false` |
| `==` | equal to | `1 == 2` | `false` |
| `!=` | not equal to | `1 != 2` | `true` |

**Note**

compare characters

You can also *compare characters*. Comparing characters is the same as comparing the Unicodes of the characters. For example, '`a`' is larger than '`A`' because the Unicode of '`a`' is larger than the Unicode of '`A`.'

**Caution**

`==` *vs.* `=`

The equality comparison operator is two equal signs (`==`), not a single equal sign (`=`). The latter symbol is for assignment.

Boolean variable

A variable that holds a Boolean value is known as a *Boolean variable*. The `boolean` data type is used to declare Boolean variables. The domain of the `boolean` type consists of two literal values: `true` and `false`. For example, the following statement assigns `true` to the variable `lightsOn`:

```
boolean lightsOn = true;
```

Boolean operators

*Boolean operators*, also known as *logical operators*, operate on Boolean values to create a new Boolean value. Table 3.2 contains a list of Boolean operators. Table 3.3 defines the not (`!`) operator. The not (`!`) operator negates `true` to `false` and `false` to `true`. Table 3.4 defines the and (`&&`) operator. The and (`&&`) of two Boolean operands is `true` if and only if both operands are `true`. Table 3.5 defines the `or` (`||`) operator. The `or` (`||`) of two Boolean operands is `true` if at least one of the operands is `true`. Table 3.6 defines the exclusive or (`^`) operator. The exclusive or (`^`) of two Boolean operands is `true` if and only if the two operands have different Boolean values.

**TABLE 3.2** Boolean Operators

| *Operator* | *Name* | *Description* |
|---|---|---|
| `!` | not | logical negation |
| `&&` | and | logical conjunction |
| `\|\|` | or | logical disjunction |
| `^` | exclusive or | logical exclusion |

**TABLE 3.3** Truth Table for Operator `!`

| *p* | *!p* | *Example* |
|---|---|---|
| `true` | `false` | `!(1 > 2)` is `true`, because `(1 > 2)` is `false`. |
| `false` | `true` | `!(1 > 0)` is `false`, because `(1 > 0)` is `true`. |

**TABLE 3.4** Truth Table for Operator `&&`

| *p1* | *p2* | *p1 && p2* | *Example* |
|---|---|---|---|
| `false` | `false` | `false` | `(2 > 3) && (5 > 5)` is `false`, because either `(2 > 3)` or `(5 > 5)` is `false`. |
| `false` | `true` | `false` | |
| `true` | `false` | `false` | `(3 > 2) && (5 > 5)` is `false`, because `(5 > 5)` is `false`. |
| `true` | `true` | `true` | `(3 > 2) && (5 >= 5)` is `true`, because `(3 > 2)` and `(5 >= 5)` are both `true`. |

**TABLE 3.5** Truth Table for Operator `||`

| *p1* | *p2* | *p1 ~p2* | *Example* |
|---|---|---|---|
| `false` | `false` | `false` | `(2 > 3) || (5 > 5)` is `false`, because `(2 > 3)` and `(5 > 5)` are both `false`. |
| `false` | `true` | `true` | |
| `true` | `false` | `true` | `(3 > 2) || (5 > 5)` is `true`, because `(3 > 2)` is `true`. |
| `true` | `true` | `true` | |

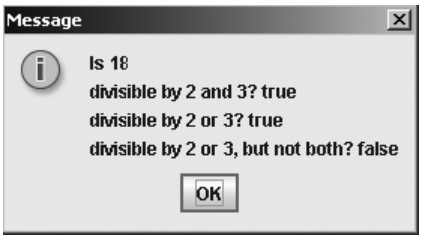
**TABLE 3.6** Truth Table for Operator `^`

| *p1* | *p2* | *p1 p2* | *Example* |
|---|---|---|---|
| `false` | `false` | `false` | `(2 > 3)^(5 > 1)` is `true`, because `(2 > 3)` is `false` and `(5 > 1)` is `true`. |
| `false` | `true` | `true` | |
| `true` | `false` | `true` | `(3 > 2)^(5 > 1)` is `false`, because both `(3 > 2)` and `(5 > 1)` are `true`. |
| `true` | `true` | `false` | |

Listing 3.1 gives a program that checks whether a number is divisible by 2 and 3, whether a number is divisible by 2 or 3, and whether a number is divisible by 2 or 3 but not both:

**LISTING 3.1** TestBoolean.java

```
1 import javax.swing.JOptionPane;
2
3 public class TestBoolean {
4   public static void main(String[] args) {
5     int number = 18;
6
7     JOptionPane.showMessageDialog(null,
8       "Is " + number +
9       "\n divisible by 2 and 3? " +
10      (number % 2 == 0 && number % 3 == 0)
11      + "\n divisible by 2 or 3? " +
12      (number % 2 == 0 || number % 3 == 0) +
13      "\n divisible by 2 or 3, but not both? "
14      + (number % 2 == 0 ^ number % 3 == 0));
15  }
16 }
```




A long string is formed by concatenating the substrings in lines 8–14. The three \n characters display the string in four lines. `(number % 2 == 0 && number % 3 == 0)` (line 10) checks whether the number is divisible by 2 and 3. `(number % 2 == 0 || number % 3 == 0)` (line 12) checks whether the number is divisible by 2 or 3. `(number % 2 == 0 ^ number % 3 == 0)` (line 14) checks whether the number is divisible by 2 or 3, but not both.

## 3.2.1 Unconditional vs. Conditional Boolean Operators

If one of the operands of an `&&` operator is `false`, the expression is `false`; if one of the operands of an `||` operand is `true`, the expression is `true`. Java uses these properties to improve the performance of these operators.

When evaluating `p1 && p2`, Java first evaluates `p1` and then evaluates `p2` if `p1` is `true`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` and then evaluates `p2` if `p1` is `false`; if `p1` is `true`, it does not evaluate `p2`. Therefore, `&&` is referred to as the *conditional* or *short-circuit AND* operator, and `||` is referred to as the *conditional* or *short-circuit OR* operator.

conditional operator
short-circuit operator

Java also provides the `&` and `|` operators. The `&` operator works exactly the same as the `&&` operator, and the `|` operator works exactly the same as the `||` operator with one exception: the `&` and `|` operators always evaluate both operands. Therefore, `&` is referred to as the *unconditional AND* operator, and `|` is referred to as the *unconditional OR* operator. In some rare situations when needed, you can use the `&` and `|` operators to guarantee that the right-hand operand is evaluated regardless of whether the left-hand operand is `true` or `false`. For example, the expression `(width < 2) & (height-- < 2)` guarantees that `(height-- < 2)` is evaluated. Thus the variable `height` will be decremented regardless of whether `width` is less than 2 or not.

unconditional operator

**Tip**

Avoid using the `&` and `|` operators. The benefits of the `&` and `|` operators are marginal. Using them will make the program difficult to read and could cause errors. For example, the expression `(x != 0) & (100 / x)` results in a runtime error if x is 0. However, `(x != 0) && (100 / x)` is fine. If `x` is `0`, `(x != 0)` is false. Since `&&` is a short-circuit operator, Java does not evaluate `(100 / x)` and returns the result as false for the entire expression `(x != 0) && (100 / x)`.

**Note**

The `&` and `|` operators can also apply to *bitwise operations*. See Appendix G, "Bit Manipulations," for details.

bitwise operations

**Note**

As shown in the preceding chapter, a `char` value can be cast into an `int` value, and vice versa. A Boolean value, however, cannot be cast into a value of other types, nor can a value of other types be cast into a Boolean value.

**Note**

`true` and `false` are literals, just like a number such as `10`, so they are not keywords, but you cannot use them as identifiers, just as you cannot use `10` as an identifier.

Boolean literals

### 3.2.2 Example: Determining Leap Year

This section presents a program that lets the user enter a year in a dialog box and checks whether it is a leap year.

A year is a *leap year* if it is divisible by `4` but not by `100` or if it is divisible by `400`. So you can use the following Boolean expression to check whether a year is a leap year:

leap year

```
(year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
```

Listing 3.2 gives the program. Two sample runs of the program are shown in Figure 3.1.

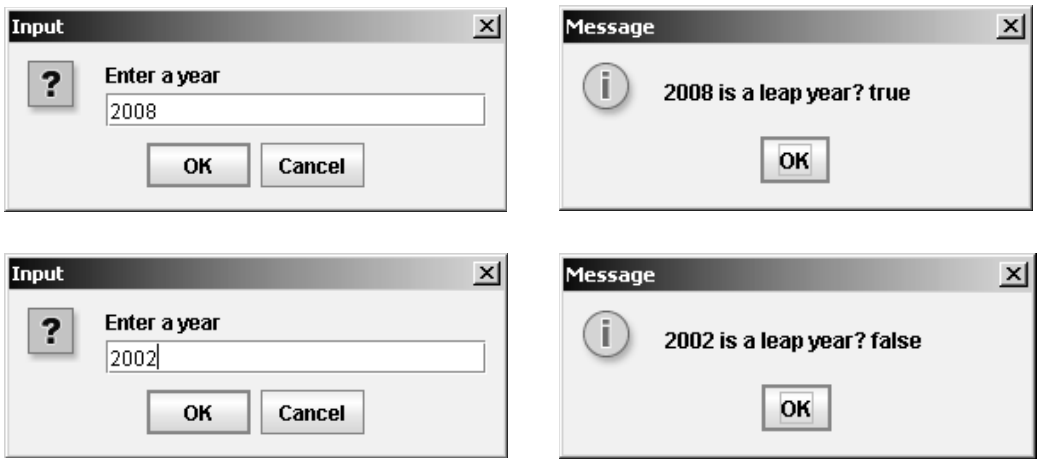



**FIGURE 3.1** The program determines whether a year is a leap year.

## LISTING 3.2 LeapYear.java

```
1 import javax.swing.JOptionPane;
2
3 public class LeapYear {
4   public static void main(String args[]) {
5     // Prompt the user to enter a year
```

show input dialog

convert to **int**

leap year?

show message dialog

```
6     String yearString = JOptionPane.showInputDialog("Enter a year");
7
8     // Convert the string into an int value
9     int year = Integer.parseInt(yearString);
10
11    // Check if the year is a leap year
12    boolean isLeapYear =
13      (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
14
15    // Display the result in a message dialog box
16    JOptionPane.showMessageDialog(null,
17      year + " is a leap year? " + isLeapYear);
18  }
19 }
```

### 3.2.3 Example: A Simple Math Learning Tool

This example creates a program to let a first grader practice addition. The program randomly generates two single-digit integers **number1** and **number2** and displays a question such as **"What is 7 + 9?"** to the student, as shown in Figure 3.2(a). After the student types the answer in the input dialog box, the program displays a message dialog box to indicate whether the answer is true or false, as shown in Figure 3.2(b).


(a) (b) (c) (d)

**FIGURE 3.2** The program generates an addition question and grades the student's answer.

There are many good ways to generate random numbers. For now, generate the first integer using **System.currentTimeMillis() % 10** and the second using **System.currentTimeMillis() * 7 % 10**. Listing 3.3 gives the program. Lines 5–6 generate two numbers, **number1** and **number2**. Line 11 displays a dialog box and obtains an answer from the user. The answer is graded in line 15 using a Boolean expression **number1 + number2 == answer**.

## LISTING 3.3 AdditionTutor.java

generate **number1**

generate **number2**

show question

display result

```
1 import javax.swing.*;
2
3 public class AdditionTutor {
4   public static void main(String[] args) {
5     int number1 = (int)(System.currentTimeMillis() % 10);
6     int number2 = (int)(System.currentTimeMillis() * 7 % 10);
7
8     String answerString = JOptionPane.showInputDialog
9       ("What is " + number1 + " + " + number2 + "?");
10
11    int answer = Integer.parseInt(answerString);
12
13    JOptionPane.showMessageDialog(null,
14      number1 + " + " + number2 + " = " + answer + " is " +
15      (number1 + number2 == answer));
16  }
17 }
```

## 3.3 if Statements

The example in Listing 3.3 displays a message such as `"6 + 2 = 7 is false."` If you wish the message to be `"6 + 2 = 7 is incorrect,"` you have to use a selection statement.

This section introduces selection statements. Java has several types of selection statements: simple `if` statements, `if ... else` statements, nested `if` statements, `switch` statements, and conditional expressions.

### 3.3.1 Simple if Statements

A simple `if` statement executes an action if and only if the condition is `true`. The syntax for a simple `if` statement is shown below:

if statement

```
if (booleanExpression) {
  statement(s);
}
```

The execution flow chart is shown in Figure 3.3(a).

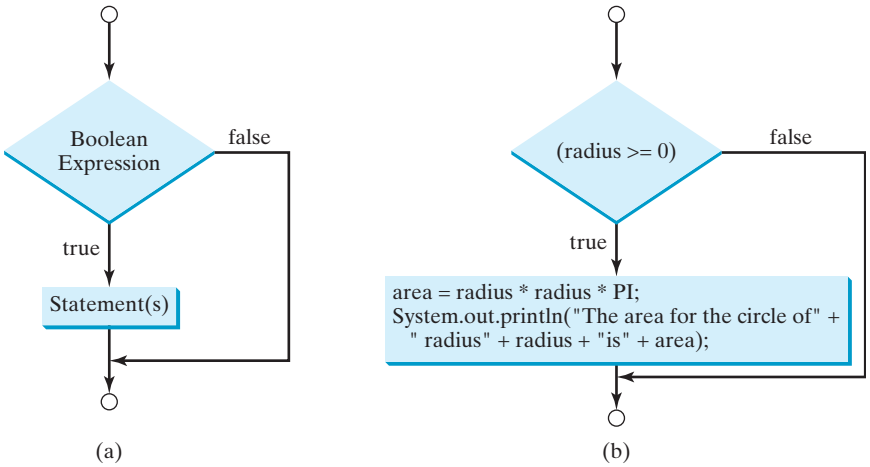



**FIGURE 3.3** An `if` statement executes statements if the `booleanExpression` evaluates to `true`.

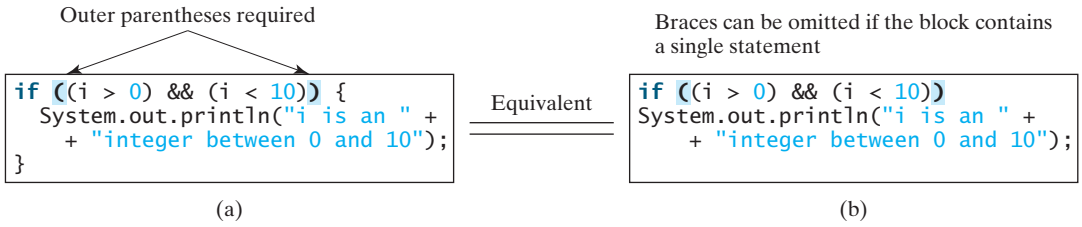
If the `booleanExpression` evaluates to `true`, the statements in the block are executed. As an example, see the following code:

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
```

The flow chart of the preceding statement is shown in Figure 3.3(b). If the value of `radius` is greater than or equal to 0, then the `area` is computed and the result is displayed; otherwise, the two statements in the block will not be executed.

**Note**

The `booleanExpression` is enclosed in parentheses for all forms of the `if` statement. Thus, for example, the outer parentheses in the following `if` statements are required.




The braces can be omitted if they enclose a single statement.

**Caution**

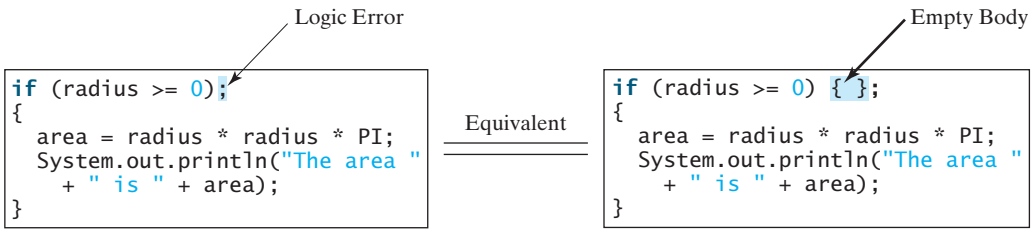
Forgetting the braces when they are needed for grouping multiple statements is a common programming error. If you modify the code by adding new statements in an `if` statement without braces, you will have to insert the braces if they are not already in place.

The following statement determines whether a number is even or odd:

```
// Prompt the user to enter an integer
String intString = JOptionPane.showInputDialog(
  "Enter an integer:");

// Convert string into int
int number = Integer.parseInt(intString);

if (number % 2 == 0)
  System.out.println(number + " is even.");

if (number % 2 != 0)
  System.out.println(number + " is odd.");
```

**Caution**

Adding a semicolon at the end of an `if` clause, as shown in (a) in the following code, is a common mistake.




This mistake is hard to find because it is neither a compilation error nor a runtime error, it is a logic error. The code in (a) is equivalent to (b) with an empty body.

This error often occurs when you use the next-line block style. Using the end-of-line block style will prevent this error.

## 3.3.2 `if ... else` Statements

A simple `if` statement takes an action if the specified condition is `true`. If the condition is `false`, nothing is done. But what if you want to take alternative actions when the condition is `false`? You can use an `if ... else` statement. The actions that an `if ... else` statement specifies differ based on whether the condition is `true` or `false`.

Here is the syntax for this type of statement:

```
if (booleanExpression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```

The flow chart of the statement is shown in Figure 3.4.

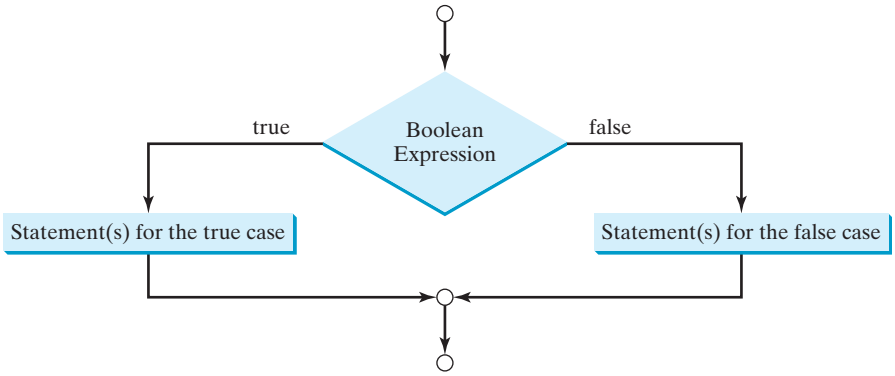



**FIGURE 3.4** An `if ... else` statement executes statements for the `true` case if the `boolean` expression evaluates to `true`; otherwise, statements for the `false` case are executed.

If the `booleanExpression` evaluates to `true`, the `statement(s)` for the true case is executed; otherwise, the `statement(s)` for the false case is executed. For example, consider the following code:

**if-else** statement

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
else {
  System.out.println("Negative input");
}
```

If `radius >= 0` is `true`, `area` is computed and displayed; if it is `false`, the message `"Negative input"` is printed.

As usual, the braces can be omitted if there is only one statement within them. The braces enclosing the `System.out.println("Negative input")` statement can therefore be omitted in the preceding example.

Using the `if … else` statement, you can rewrite the code for determining whether a number is even or odd in the preceding section, as follows:

```
if (number % 2 == 0)
  System.out.println(number + " is even.");
else
  System.out.println(number + " is odd.");
```

This is more efficient because whether `number % 2` is `0` is tested only once.

### 3.3.3 Nested `if` Statements

The statement in an `if` or `if ... else` statement can be any legal Java statement, including another `if` or `if ... else` statement. The inner `if` statement is said to be *nested* inside the outer `if` statement. The inner `if` statement can contain another `if` statement; in fact, there is no limit to the depth of the nesting. For example, the following is a nested `if` statement:

nested `if` statement

```
if (i > k) {
  if (j > k)
    System.out.println("i and j are greater than k");
}
else
  System.out.println("i is less than or equal to k");
```

The `if (j > k)` statement is nested inside the `if (i > k)` statement.

The nested `if` statement can be used to implement multiple alternatives. The statement given in Figure 3.5(a), for instance, assigns a letter grade to the variable `grade` according to the score, with multiple alternatives.

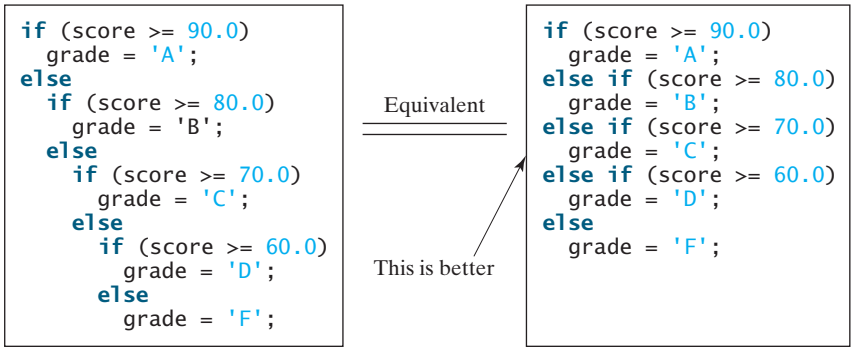



**FIGURE 3.5** A preferred format for multiple alternative `if` statements is shown in (b).

The execution of this `if` statement proceeds as follows. The first condition `(score >= 90.0)` is tested. If it is `true`, the grade becomes `'A'`. If it is `false`, the second condition `(score >= 80.0)` is tested. If the second condition is `true`, the grade becomes `'B'`. If that condition is `false`, the third condition and the rest of the conditions (if necessary) continue to be tested until a condition is met or all of the conditions prove to be `false`. If all of the conditions are `false`, the grade becomes `'F'`. Note that a condition is tested only when all of the conditions that come before it are `false`.

The `if` statement in Figure 3.5(a) is equivalent to the `if` statement in Figure 3.5(b). In fact, Figure 3.5(b) is the preferred writing style for multiple alternative `if` statements. This style avoids deep indentation and makes the program easy to read.

**Note**

matching `else` with `if`

The `else` clause matches the most recent unmatched `if` clause in the same block. For example, the following statement in (a) is equivalent to the statement in (b).

The compiler ignores indentation. Nothing is printed from the statements in (a) and (b). To force the `else` clause to match the first `if` clause, you must add a pair of braces:

```
int i = 1; int j = 2; int k = 3;
if (i > j) {
```

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
   if (i > k)
      System.out.println("A");
else
      System.out.println("B");
```

(a)

Equivalent

This is better with correct indentation

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
   if (i > k)
      System.out.println("A");
   else
      System.out.println("B");
```

(b)

```
  if (i > k)
    System.out.println("A");
}
else
  System.out.println("B");
```

This statement prints **B**.

## Tip

Often new programmers write the code that assigns a test condition to a **boolean** variable like the code in (a):

assign boolean variable

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

Equivalent

This is better

```
boolean even
  = number % 2 == 0;
```

(b)

The code can be simplified by assigning the test value directly to the variable, as shown in (b).

## Caution

To test whether a **boolean** variable is **true** or **false** in a test condition, it is redundant to use the equality comparison operator like the code in (a):

test boolean value

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

This is better

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

Instead, it is better to use the **boolean** variable directly, as shown in (b). Another good reason to use the **boolean** variable directly is to avoid errors that are difficult to detect. Using the = operator instead of the **==** operator to compare equality of two items in a test condition is a common error. It could lead to the following erroneous statement:

```
if (even = true)
  System.out.println("It is even.");
```

This statement does not have syntax errors. It assigns **true** to **even** so that **even** is always **true**.

debugging in IDE

**Tip**

If you use an IDE such as JBuilder, NetBeans, or Eclipse, please refer to *Learning Java Effectively with JBuilder/NetBeans/Eclipse* in the supplements. This supplement shows you how to use a debugger to trace a simple `if-else` statement.

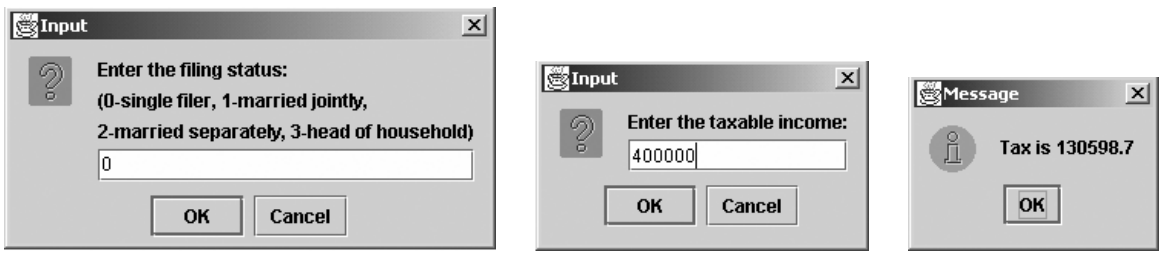
### 3.3.4 Example: Computing Taxes

This section uses nested `if` statements to write a program to compute personal income tax. The United States federal personal income tax is calculated based on filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates for 2002 are shown in Table 3.7. If you are, say, single with a taxable income of $10,000, the first $6,000 is taxed at 10% and the other $4,000 is taxed at 15%. So your tax is $1,200.

**TABLE 3.7** 2002 U.S. Federal Personal Tax Rates

| *Tax rate* | *Single filers* | *Married filing jointly or qualifying widow/widower* | *Married filing separately* | *Head of household* |
|---|---|---|---|---|
| 10% | Up to $6,000 | Up to $12,000 | Up to $6,000 | Up to $10,000 |
| 15% | $6,001–$27,950 | $12,001–$46,700 | $6,001–$23,350 | $10,001–$37,450 |
| 27% | $27,951–$67,700 | $46,701–$112,850 | $23,351–$56,425 | $37,451–$96,700 |
| 30% | $67,701–$141,250 | $112,851–$171,950 | $56,426–$85,975 | $96,701–$156,600 |
| 35% | $141,251–$307,050 | $171,951–$307,050 | $85,976–$153,525 | $156,601–$307,050 |
| 38.6% | $307,051 or more | $307,051 or more | $153,526 or more | $307,051 or more |

Your program should prompt the user to enter the filing status and taxable income and computes the tax for the year 2002. Enter 0 for single filers, 1 for married filing jointly, 2 for married filing separately, and 3 for head of household. A sample run of the program is shown in Figure 3.6.




(a) (b) (c)

**FIGURE 3.6** The program computes the tax using `if` statements.

Your program computes the tax for the taxable income based on the filing status. The filing status can be determined using `if` statements outlined as follows:

```
if (status == 0) {
  // Compute tax for single filers
}
else if (status == 1) {
  // Compute tax for married file jointly
}
else if (status == 2) {
  // Compute tax for married file separately
}
```

```
else if (status == 3) {
  // Compute tax for head of household
}
else {
  // Display wrong status
}
```

For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, of a taxable income of $400,000 for single filers, $6,000 is taxed at 10%, (27950 – 6000) at 15%, (67700 – 27950) at 27%, (141250 – 67700) at 35%, and (400000 – 307050) at 38.6%.

Listing 3.4 gives the solution to compute taxes for single filers. The complete solution is left as an exercise.

## LISTING 3.4 ComputeTaxWithSelectionStatement.java

```
1  import javax.swing.JOptionPane;                                    import class
2
3  public class ComputeTaxWithSelectionStatement {
4    public static void main(String[] args) {
5      // Prompt the user to enter filing status
6      String statusString = JOptionPane.showInputDialog(            input dialog
7        "Enter the filing status:\n" +
8        "(0-single filer, 1-married jointly,\n" +
9        "2-married separately, 3-head of household)");
10     int status = Integer.parseInt(statusString);                   convert string
11                                                                      to int
12     // Prompt the user to enter taxable income
13     String incomeString = JOptionPane.showInputDialog(             input dialog
14       "Enter the taxable income:");
15     double income = Double.parseDouble(incomeString);              convert string
16                                                                      to double
17     // Compute tax
18     double tax = 0;                                                compute tax
19
20     if (status == 0) { // Compute tax for single filers
21       if (income <= 6000)
22         tax = income * 0.10;
23       else if (income <= 27950)
24         tax = 6000 * 0.10 + (income - 6000) * 0.15;
25       else if (income <= 67700)
26         tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
27           (income - 27950) * 0.27;
28       else if (income <= 141250)
29         tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
30           (67700 - 27950) * 0.27 + (income - 67700) * 0.30;
31       else if (income <= 307050)
32         tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
33           (67700 - 27950) * 0.27 + (141250 - 67700) * 0.30 +
34           (income - 141250) * 0.35;
35       else
36         tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
37           (67700 - 27950) * 0.27 + (141250 - 67700) * 0.30 +
38           (307050 - 141250) * 0.35 + (income - 307050) * 0.386;
39     }
40     else if (status == 1) { // Compute tax for married file jointly
41       // Left as exercise
42     }
```

```
43      else if (status == 2) { // Compute tax for married separately
44        // Left as exercise
45      }
46      else if (status == 3) { // Compute tax for head of household
47        // Left as exercise
48      }
49      else {
50        System.out.println("Error: invalid status");
51        System.exit(0);
52      }
53
54      // Display the result
55      JOptionPane.showMessageDialog(null, "Tax is " +
56        (int)(tax * 100) / 100.0);
57    }
58  }
```

exit program

message dialog

The `import` statement (line 1) makes the class `javax.swing.JOptionPane` available for use in this example.

The program receives the filing status and taxable income. The multiple alternative `if` statements (lines 22, 42, 45, 48, 51) check the filing status and compute the tax based on the filing status.

Like the `showMessageDialog` method, `System.exit(0)` (line 53) is also a static method. This method is defined in the `System` class. Invoking this method terminates the program. The argument `0` indicates that the program is terminated normally.

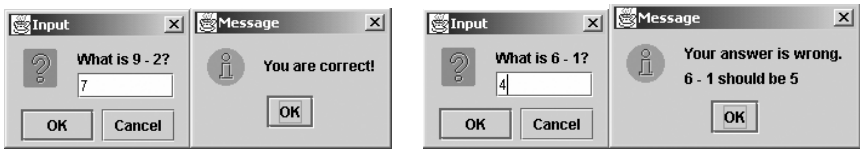
exit method

**Note**

An initial value of `0` is assigned to `tax` (line 20). A syntax error would occur if it had no initial value because all of the other statements that assign values to `tax` are within the `if` statement. The compiler thinks that these statements may not be executed and therefore reports a syntax error.

### 3.3.5 Example: An Improved Math Learning Tool

This example creates a program for a first grader to practice subtraction. The program randomly generates two single-digit integers `number1` and `number2` with `number1 > number2` and displays a question such as "`What is 9-2?`" to the student, as shown in Figure 3.7(a). After the student types the answer in the input dialog box, the program displays a message dialog box to indicate whether the answer is correct, as shown in Figure 3.7(b).




(a) (b) (c) (d)

**FIGURE 3.7** The program generates a subtraction question and grades the student's answer.

`random()` method

To generate a random number, use the `random()` method in the `Math` class. Invoking this method returns a random double value `d` such that `0.0 ≤ d < 1.0` So `(int)(Math.random() * 10)` returns a random single-digit integer (i.e., a number between `0` and `9`).

The program may work as follows:

- Generate two single-digit integers into `number1` and `number2`.
- If `number1 < number2`, swap `number1` with `number2`.

- Prompt the student to answer "what is number1 – number2?"
- Check the student's answer and display whether the answer is correct.

The complete program is shown in Listing 3.5.

### LISTING 3.5 SubtractionTutor.java

```
1  import javax.swing.JOptionPane;                                    import class
2
3  public class SubtractionTutor {
4    public static void main(String[] args) {
5      // 1. Generate two random single-digit integers
6      int number1 = (int)(Math.random() * 10);                       random numbers
7      int number2 = (int)(Math.random() * 10);
8
9      // 2. If number1 < number2, swap number1 with number2
10     if (number1 < number2) {
11       int temp = number1;
12       number1 = number2;
13       number2 = temp;
14     }
15
16     // 3. Prompt the student to answer "what is number1 - number2?"
17     String answerString = JOptionPane.showInputDialog               input dialog
18       ("What is " + number1 + " - " + number2 + "?");
19     int answer = Integer.parseInt(answerString);
20
21     // 4. Grade the answer and display the result
22     String replyString;
23     if (number1 - number2 == answer)
24       replyString = "You are correct!";
25     else
26       replyString = "Your answer is wrong.\n" + number1 + " - "
27         + number2 + " should be " + (number1 - number2);
28     JOptionPane.showMessageDialog(null, replyString);               message dialog
29   }
30 }
```

To swap two variables `number1` and `number2`, a temporary variable `temp` (line 11) is used to first hold the value in `number1`. The value in `number2` is assigned to `number1` (line 12), and the value in `temp` is assigned to `number2`.

## 3.4 switch Statements

The `if` statement in Listing 3.4 makes selections based on a single `true` or `false` condition. There are four cases for computing taxes, which depend on the value of `status`. To fully account for all the cases, nested `if` statements were used. Overuse of nested `if` statements makes a program difficult to read. Java provides a `switch` statement to handle multiple conditions efficiently. You could write the following `switch` statement to replace the nested `if` statement in Listing 3.4:

```
switch (status) {
  case 0:  compute taxes for single filers;
           break;
  case 1:  compute taxes for married file jointly;
           break;
```

```
    case 2:  compute taxes for married file separately;
             break;
    case 3:  compute taxes for head of household;
             break;
    default: System.out.println("Errors: invalid status");
             System.exit(0);
  }
```

The flow chart of the preceding `switch` statement is shown in Figure 3.8.

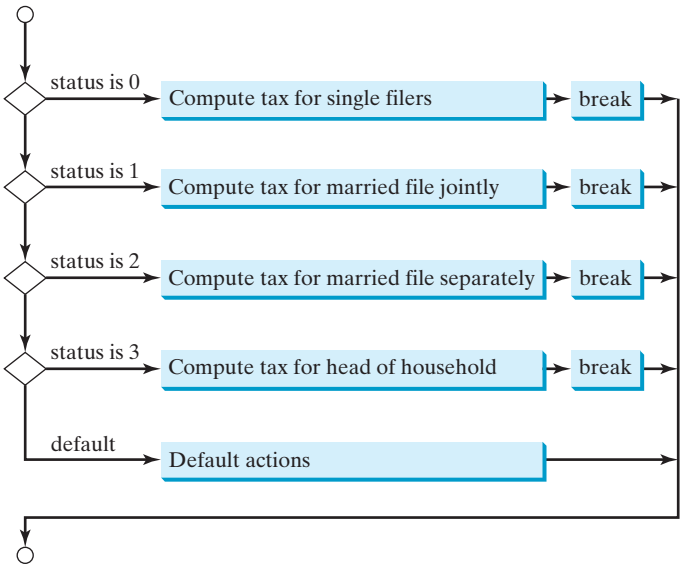



**FIGURE 3.8** The `switch` statement checks all cases and executes the statements in the matched case.

This statement checks to see whether the status matches the value `0`, `1`, `2`, or `3`, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the `switch` statement:

`switch` statement

```
switch (switch-expression) {
  case value1: statement(s)1;
               break;
  case value2: statement(s)2;
               break;
  ...
  case valueN: statement(s)N;
               break;
  default:     statement(s)-for-default;
}
```

The `switch` statement observes the following rules:

- The `switch-expression` must yield a value of `char`, `byte`, `short`, or `int` type and must always be enclosed in parentheses.
- The `value1`, ..., and `valueN` must have the same data type as the value of the `switch-expression`. Note that `value1`, ..., and `valueN` are constant expressions, meaning that they cannot contain variables in the expression, such as `1 + x`.

- When the value in a `case` statement matches the value of the `switch-expression`, the statements *starting from this case* are executed until either a `break` statement or the end of the switch statement is reached.
- The keyword `break` is optional. The `break` statement immediately ends the `switch` statement.
- The `default` case, which is optional, can be used to perform actions when none of the specified cases matches the `switch-expression`.
- The `case` statements are checked in sequential order, but the order of the cases (including the default case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.

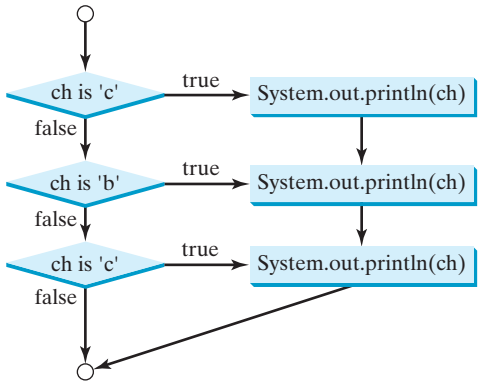
### Caution

without **break**

Do not forget to use a `break` statement when one is needed. Once a case is matched, the statements starting from the matched case are executed until a `break` statement or the end of the `switch` statement is reached. This phenomenon is referred to as the *fall-through* behavior. For example, the following code prints character a three times if `ch` is `'a'`:

fall-through behavior

```
switch (ch) {
  case 'a': System.out.println(ch);
  case 'b': System.out.println(ch);
  case 'c': System.out.println(ch);
}
```




### Tip

To avoid programming errors and improve code maintainability, it is a good idea to put a comment in a case clause if `break` is purposely omitted.

## 3.5 Conditional Expressions

You might want to assign a value to a variable that is restricted by certain conditions. For example, the following statement assigns `1` to `y` if `x` is greater than `0`, and `-1` to `y` if `x` is less than or equal to `0`.

```
if (x > 0)
  y = 1
else
  y = -1;
```

Alternatively, as in this example, you can use a conditional expression to achieve the same result.

```
y = (x > 0) ? 1 : -1;
```

Conditional expressions are in a completely different style, with no explicit `if` in the statement. The syntax is shown below:

```
booleanExpression ? expression1 : expression2;
```

conditional expression

The result of this conditional expression is `expression1` if `booleanExpression` is `true`; otherwise the result is `expression2`.

Suppose you want to assign the larger number between variable `num1` and `num2` to `max`. You can simply write a statement using the conditional expression:

```
max = (num1 > num2) ? num1 : num2;
```

For another example, the following statement displays the message `"num is even"` if `num` is even, and otherwise displays `"num is odd."`

```
System.out.println((num % 2 == 0) ? "num is even" : "num is odd");
```

**Note**

The symbols `?` and `:` appear together in a conditional expression. They form a conditional operator. This operator is called a *ternary operator* because it uses three operands. It is the only ternary operator in Java.

## 3.6 Formatting Console Output and Strings

You already know how to display console output using the `println` method. JDK 1.5 introduced a new `printf` method that enables you to format output. The syntax to invoke this method is

`printf`

```
System.out.printf(format, item1, item2, ..., itemk)
```

specifier

where `format` is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, a character, a boolean value, or a string. Each specifier begins with a percent sign. Table 3.8 lists some frequently used specifiers.

**TABLE 3.8** Frequently Used Specifiers

| *Specifier* | *Output* | *Example* |
|---|---|---|
| `%b` | a boolean value | `true` or `false` |
| `%c` | a character | `'a'` |
| `%d` | a decimal integer | `200` |
| `%f` | a floating-point number | `45.460000` |
| `%e` | a number in standard scientific notation | `4.556000e+01` |
| `%s` | a string | `"Java is cool"` |

Here is an example:

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

items

```
display              count is 5 and amount is 45.560000
```

Items must match the specifiers in order, in number, and in exact type. For example, the specifier for `count` is `%d` and for `amount` is `%f`. By default, a floating-point value is displayed with six digits after the decimal point. You can specify the width and precision in a specifier, as shown in the examples in Table 3.9.

**TABLE 3.9** Examples of Specifying Width and Precision

| *Example* | *Output* |
|---|---|
| `%5c` | Output the character and add four spaces before the character item. |
| `%6b` | Output the boolean value and add one space before the false value and two spaces before the true value. |
| `%5d` | Output the integer item with width at least `5`. If the number of digits in the item is `<5`, add spaces before the number. If the number of digits in the item is `>5`, the width is automatically increased. |
| `%10.2f` | Output the floating-point item with width at least `10` including a decimal point and two digits after the point. Thus there are `7` digits allocated before the decimal point. If the number of digits before the decimal in the item is `<7`, add spaces before the number. If the number of digits before the decimal in the item is `>7`, the width is automatically increased. |
| `%10.2e` | Output the floating-point item with width at least `10` including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than `10`, add spaces before the number. |
| `%12s` | Output the string with width at least `12` characters. If the string item has less than `12` characters, add spaces before the string. If the string item has more than `12` characters, the width is automatically increased. |

left justify

You can put the minus sign (`-`) in the specifier to specify that the item is left-justified in the output within the specified field. For example, the following statement

```
System.out.printf("%8d%-8s\n", 1234, "Java");
System.out.printf("%-8d%-8s\n", 1234, "Java");
```

displays

```
    1234Java
1234    Java
```

**Caution**

The items must match the specifiers in exact type. The item for the specifier `%f` or `%e` must be a floating-point type value such as `40.0`, not `40`. Thus an `int` variable cannot match `%f` or `%e`.

**Tip**

The `%` sign denotes a specifier. To output a literal `%` in the format string, use `%%`.

formatting strings

You can print formatted output to the console using the `printf` method. Can you display formatted output in a message dialog box? To accomplish this, use the static `format` method in the `String` class to create a formatted string. The syntax to invoke this method is

```
String.format(format, item1, item2, ..., itemk)
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```
String s = String.format("count is %d and amount is %f", 5, 45.56));
```

creates a formatted string `"count is 5 and amount is 45.560000"`.

The following statement displays a formatted string in a message dialog box:

```
JOptionPane.showMessageDialog(null,
  String.format("Sales tax is %1.2f",
    24.3454));
```


## 3.7 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated. Suppose that you have this expression:

```
3 + 4 * 4 > 5 * (4 + 3) - 1
```

What is its value? How does the compiler know the execution order of the operators? The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule. The precedence rule defines precedence for operators, as shown in Table 3.10, which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. Operators with the same precedence appear in the same group. (See Appendix C, "Operator Precedence Chart," for a complete list of Java operators and their precedence.)

precedence

**TABLE 3.10** Operator Precedence Chart

| *Precedence* | *Operator* |
|---|---|
| Highest Order | `var++` and `var--` (Postfix) |
| | `+`, `-` (Unary plus and minus), `++var` and `--var` (Prefix) |
| | (type) (Casting) |
| | `!` (Not) |
| | `*`, `/`, `%` (Multiplication, division, and remainder) |
| | `+`, `-` (Binary addition and subtraction) |
| | `<`, `<=`, `>`, `>=` (Comparison) |
| | `==`, `!=` (Equality) |
| | `&` (Unconditional AND) |
| | `^` (Exclusive OR) |
| | `\|` (Unconditional OR) |
| | `&&` (Conditional AND) |
| | `\|\|` (Conditional OR) |
| Lowest Order | `=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operator) |

associativity

If operators with the same precedence are next to each other, their *associativity* determines the order of evaluation. All binary operators except assignment operators are *left-associative*. For example, since + and − are of the same precedence and are left-associative, the expression

```
a - b + c - d   ——equivalent——   ((a - b) + c) - d
```

Assignment operators are *right-associative*. Therefore, the expression

```
a = b += c = 5   ═══ equivalent ═══   a = (b += (c = 5))
```

Suppose `a`, `b`, and `c` are `1` before the assignment; after the whole expression is evaluated, `a` becomes `6`, `b` becomes `6`, and `c` becomes `5`. Note that left associativity for the assignment operator would not make sense.

Applying the operator precedence and associativity rule, the expression `3 + 4 * 4 > 5 * (4 + 3) - 1` is evaluated as follows:

```
3 + 4 * 4 > 5 * (4 + 3) - 1
                (1) inside parentheses first
3 + 4 * 4 > 5 * 7 - 1
            (2) multiplication
3 + 16 > 5 * 7 - 1
         (3) multiplication
3 + 16 > 35 - 1
       (4) addition
19 > 35 - 1
     (5) subtraction
19 > 34
   (6) greater than
false
```

**Tip**

You can use parentheses to force an evaluation order as well as to make a program easy to read. Use of redundant parentheses does not slow down the execution of the expression.

## 3.8 Operand Evaluation Order

The precedence and associativity rules specify the order of the operators but not the order in which the operands of a binary operator are evaluated. Operands are evaluated strictly *from left to right* in Java. *The left-hand operand of a binary operator is evaluated before any part of the right-hand operand is evaluated.* This rule takes precedence over any other rules that govern expressions. Consider this expression:

from left to right

```
a + b * (c + 10 * d) / e
```

`a`, `b`, `c`, `d`, and `e` are evaluated in this order. *If no operands have side effects that change the value of a variable, the order of operand evaluation is irrelevant.* Interesting cases arise when operands do have a side effect. For example, `x` becomes 1 in the following code because `a` is evaluated to 0 before `++a` is evaluated to `1`.

```
int a = 0;
int x = a + (++a);
```

But `x` becomes `2` in the following code because `++a` is evaluated to `1`, and then `a` is evaluated to `1`.

```
int a = 0;
int x = ++a + a;
```

*The order for evaluating operands takes precedence over the operator precedence rule.* In the former case, `(++a)` has higher precedence than addition (`+`), but since `a` is a left-hand

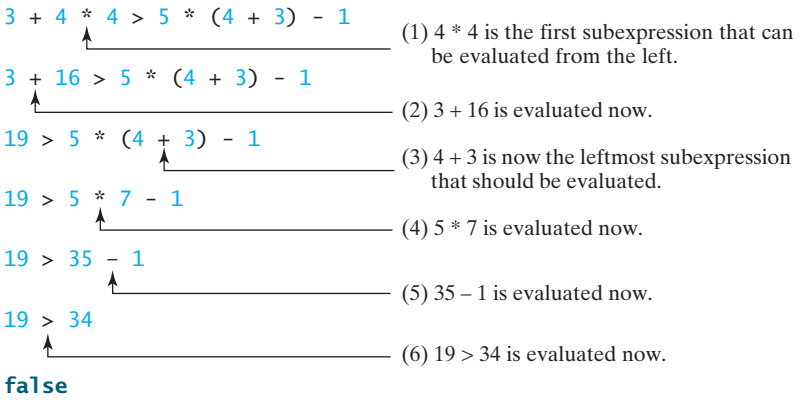
operand of the addition (+), it is evaluated before any part of its right-hand operand (e.g., ++a in this case).

evaluation rule

In summary, the rule of evaluating an expression is:

- Rule 1: Evaluate whatever subexpressions you can possibly evaluate from left to right.
- Rule 2: The operators are applied according to their precedence, as shown in Table 3.10.
- Rule 3: The associativity rule applies for two operators next to each other with the same precedence.

Applying the rule, the expression `3 + 4 * 4 > 5 * (4 + 3) - 1` is evaluated as follows:




The result happens to be the same as applying Rule 2 and Rule 3 without applying Rule 1. In fact, Rule 1 is not necessary if no operands have side effects that change the value of a variable in an expression.

## Key Terms

boolean expression 68
boolean value 68
`boolean` type 68
`break` statement 81, 101
conditional operator 84
fall-through behavior 83
operand evaluation order 87
operator associativity 86
operator precedence 86
selection statement 73
short-circuit evaluation 70

## Chapter Summary

- Java has eight primitive data types. The preceding chapter introduced `byte`, `short`, `int`, `long`, `float`, `double`, and `char`. This chapter introduced the `boolean` type that represents a `true` or `false` value.
- The Boolean operators `&&`, `&`, `||`, `|`, `!`, and `^` operate with Boolean values and variables. The relational operators (`<`, `<=`, `==`, `!=`, `>`, `>=`) work with numbers and characters, and yield a Boolean value.
- When evaluating `p1 && p2`, Java first evaluates `p1` and then evaluates `p2` if `p1` is `true`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` and then evaluates `p2` if `p1` is `false`; if `p1` is `true`, it does not evaluate

`p2`. Therefore, `&&` is referred to as the *conditional* or *short-circuit AND* operator, and `||` is referred to as the *conditional* or *short-circuit OR* operator.

- Java also provides the `&` and `|` operators. The `&` operator works exactly the same as the `&&` operator, and the `|` operator works exactly the same as the `||` operator with one exception: the `&` and `|` operators always evaluate both operands. Therefore, `&` is referred to as the *unconditional AND* operator, and `|` is referred to as the *unconditional OR* operator.

- Selection statements are used for building selection steps into programs. There are several types of selection statements: `if` statements, `if ... else` statements, nested `if` statements, `switch` statements, and conditional expressions.

- The various `if` statements all make control decisions based on a Boolean expression. Based on the `true` or `false` evaluation of the expression, these statements take one of two possible courses.

- The `switch` statement makes control decisions based on a switch expression of type `char`, `byte`, `short`, `int`, or `boolean`.

- The keyword `break` is optional in a `switch` statement, but it is normally used at the end of each case in order to terminate the remainder of the `switch` statement. If the `break` statement is not present, the next `case` statement will be executed.

- The operands of a binary operator are evaluated from left to right. No part of the right-hand operand is evaluated until all the operands before the binary operator are evaluated.

- The operators in arithmetic expressions are evaluated in the order determined by the rules of parentheses, operator precedence, and associativity.

- Parentheses can be used to force the order of evaluation to occur in any sequence. Operators with higher precedence are evaluated earlier. The associativity of the operators determines the order of evaluation for operators of the same precedence.

- All binary operators except assignment operators are left-associative, and assignment operators are right-associative.

## Review Questions

### Section 3.2 `boolean` Data Type and Operations

**3.1** List six comparison operators.

**3.2** Assume that `x` is `1`, show the result of the following Boolean expressions:

```
(true) && (3 > 4)
!(x > 0) && (x > 0)
(x > 0) || (x < 0)
(x != 0) || (x == 0)
(x >= 0) || (x < 0)
(x != 1) == !(x == 1)
```

**3.3** Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100`.

**3.4** Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100` or the number is negative.

**3.5** Assume that `x` and `y` are `int` type. Which of the following are correct Java expressions?

```
x > y > 0
x = y && y
x /= y
x or y
x and y
(x != 0) || (x = 0)
```

**3.6** Can the following conversions involving casting be allowed? If so, find the converted result.

```
boolean b = true;
i = (int)b;
int i = 1;
boolean b = (boolean)i;
```

**3.7** Suppose that `x` is 1. What is `x` after the evaluation of the following expression?

```
(x > 1) & (x++ > 1)
```

**3.8** Suppose that `x` is 1. What is `x` after the evaluation of the following expression?

```
(x > 1) && (x++ > 1)
```

**3.9** Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    char x = 'a';
    char y = 'c';

    System.out.println(++y);
    System.out.println(y++);
    System.out.println(x > y);
    System.out.println(x - y);
  }
}
```

## Section 3.3 `if` Statements

**3.10** Suppose `x = 3` and `y = 2`, show the output, if any, of the following code. What is the output if `x = 3` and `y = 4`? What is the output if `x = 2` and `y = 2`? Draw a flowchart of the following code:

```
if (x > 2) {
  if (y > 2) {
    int z = x + y;
    System.out.println("z is " + z);
  }
}
else
  System.out.println("x is " + x);
```

**3.11** Which of the following statements are equivalent? Which ones are correctly indented?

```
if (i > 0) if
(j > 0)
x = 0; else
if (k > 0) y = 0;
else z = 0;
```

(a)

```
if (i > 0) {
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
}
else
  z = 0;
```

(b)

```
if (i > 0)
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
  else
    z = 0;
```

(c)

```
if (i > 0)
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
else
  z = 0;
```

(d)

**3.12** Suppose `x = 2` and `y = 3`, show the output, if any, of the following code. What is the output if `x = 3` and `y = 2`? What is the output if `x = 3` and `y = 3`? (Hint: please indent the statement correctly first.)

```
if (x > 2)
  if (y > 2) {
    int z = x + y;
    System.out.println("z is " + z);
  }
else
  System.out.println("x is " + x);
```

**3.13** Are the following two statements equivalent?

```
if (income <= 10000)
  tax = income * 0.1;
else if (income <= 20000)
  tax = 1000 +
   (income - 10000) * 0.15;
```

```
if (income <= 10000)
  tax = income * 0.1;
else if (income > 10000 &&
         income <= 20000)
  tax = 1000 +
   (income - 10000) * 0.15;
```

**3.14** Which of the following is a possible output from invoking `Math.random()`?

`323.4`, `0.5`, `34`, `1.0`, `0.0`, `0.234`

**3.15** How do you generate a random integer `i` such that $0 \le i < 20$?

How do you generate a random integer `i` such that $10 \le i < 20$?

How do you generate a random integer `i` such that $10 \le i \le 50$?

## Section 3.4 `switch` Statements

**3.16** What data types are required for a `switch` variable? If the keyword `break` is not used after a case is processed, what is the next statement to be executed? Can you convert a `switch` statement to an equivalent `if` statement, or vice versa? What are the advantages of using a `switch` statement?

**3.17** What is `y` after the following `switch` statement is executed?

```
x = 3; y = 3;
switch (x + 3) {
  case 6:  y = 1;
  default: y += 1;
}
```

**3.18** Use a `switch` statement to rewrite the following `if` statement and draw the flowchart for the `switch` statement:

```
if (a == 1)
  x += 5;
else if (a == 2)
  x += 10;
else if (a == 3)
  x += 16;
else if (a == 4)
  x += 34;
```

## Section 3.5 Conditional Expressions

**3.19** Rewrite the following `if` statement using the conditional operator:

```
if (count % 10 == 0)
  System.out.print(count + "\n");
else
  System.out.print(count + " ");
```

## Section 3.6 Formatting Console Output and Strings

**3.20** What are the specifiers for outputting a boolean value, a character, a decimal integer, a floating-point number, and a string?

**3.21** What is wrong in the following statements?

```
(a) System.out.printf("%5d %d", 1, 2, 3);
(b) System.out.printf("%5d %f", 1);
(c) System.out.printf("%5d %f", 1, 2);
```

**3.22** Show the output of the following statements.

```
(a) System.out.printf("amount is %f %e\n", 32.32, 32.32);
(b) System.out.printf("amount is %5.4f %5.4e\n", 32.32, 32.32);
(c) System.out.printf("%6b\n", (1 > 2));
(d) System.out.printf("%6s\n", "Java");
(e) System.out.printf("%-6b%s\n", (1 > 2), "Java");
(f) System.out.printf("%6b%-s\n", (1 > 2), "Java");
```

**3.23** How do you create a formatted string?

## Sections 3.7–3.8

**3.24** List the precedence order of the Boolean operators. Evaluate the following expressions:

```
true | true && false
true || true && false
true | true & false
```

**3.25** Show and explain the output of the following code:

(a)
```
int i = 0;
System.out.println(--i + i + i++);
System.out.println(i + ++i);
```

(b)
```
int i = 0;
i = i + (i = 1);
System.out.println(i);
```

(c)
```
int i = 0;
i = (i = 1) + i;
System.out.println(i);
```

**3.26** Assume that `int a = 1` and `double d = 1.0`, and that each expression is independent. What are the results of the following expressions?

```
a = (a = 3) + a;
a = a + (a = 3);
a += a + (a = 3);
a = 5 + 5 * 2 % a--;
a = 4 + 1 + 4 * 5 % (++a + 1);
d += 1.5 * 3 + (++d);
d -= 1.5 * 3 + d++;
```

## Programming Exercises

### Section 3.2 `boolean` Data Type and Operations

**Pedagogical Note**

For each exercise, students should carefully analyze the problem requirements and the design strategies for solving the problem before coding.

think before coding

**Pedagogical Note**

Instructors may ask students to *document analysis and design* for selected exercises. Students should use their own words to analyze the problem, including the input, output, and what needs to be computed and describe how to solve the problem using pseudocode. This has two benefits: (1) it mandates students to think before typing code; (2) it fosters writing skills.

document analysis and design

**Debugging Tip**

Before you ask for help, read and explain the program to yourself, and trace it using several representative inputs by hand or using an IDE debugger. You learn how to program by debugging your own mistakes.

learn from mistakes

**Note**

Do not use selection statements for Exercises 3.1–3.6.

**3.1*** (*Validating triangles*) Write a program that reads three edges for a triangle and determines whether the input is valid. The input is valid if the sum of any two edges is greater than the third edge. For example, if your input for three edges is 1, 2, 1, the output should be:

```
Can edges 1, 2, and 1 form a triangle? false
```

If your input for three edges is 2, 2, 1, the output should be:

```
Can edges 2, 2, and 1 form a triangle? true
```

**3.2** (*Checking whether a number is even*) Write a program that reads an integer and checks whether it is even. For example, if your input is 25, the output should be:

```
Is 25 an even number? false
```

If your input is 2000, the output should be:

```
Is 2000 an even number? true
```

**3.3*** (*Using the* `&&`, `||` *and* `^` *operators*) Write a program that prompts the user to enter an integer and determines whether it is divisible by `5` and `6`, whether it is divisible by `5` or `6`, and whether it is divisible by `5` or `6`, but not both. For example, if your input is `10`, the output should be

```
Is 10 divisible by 5 and 6? false
Is 10 divisible by 5 or 6? true
Is 10 divisible by 5 or 6, but not both? true
```

**3.4**** (*Learning addition*) Write a program that generates two integers under `100` and prompts the user to enter the addition of these two integers. The program then reports `true` if the answer is correct, `false` otherwise. The program is similar to Listing 3.3.

**3.5**** (*Addition for three numbers*) The program in Listing 3.3 generates two integers and prompts the user to enter the addition of these two integers. Revise the program to generate three single-digit integers and prompt the user to enter the addition of these three integers.

**3.6*** (*Using the console input*) Rewrite Listing 3.2, LeapYear.java, using the console input.

## Section 3.3 Selection Statements

**3.7** (*Monetary units*) Modify Listing 2.7 to display the non-zero denominations only, using singular words for single units like `1` dollar and `1` penny, and plural words for more than one unit like `2` dollars and `3` pennies. (Use `23.67` to test your program.)

**3.8*** (*Sorting three integers*) Write a program that sorts three integers. The integers are entered from the input dialogs and stored in variables `num1`, `num2`, and `num3`, respectively. The program sorts the numbers so that $num1 \le num2 \le num3$.

**3.9** (*Computing the perimeter of a triangle*) Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid. The input is valid if the sum of any two edges is greater than the third edge (also see Exercise 3.1).

**3.10** (*Computing taxes*) Listing 3.4 gives the source code to compute taxes for single filers. Complete Listing 3.4 to give the complete source code.

**3.11*** (*Finding the number of days in a month*) Write a program that prompts the user to enter the month and year, and displays the number of days in the month. For example, if the user entered month 2 and year 2000, the program should display that February 2000 has 29 days. If the user entered month 3 and year 2005, the program should display that March 2005 has 31 days.

**3.12** (*Checking a number*) Write a program that prompts the user to enter an integer and checks whether the number is divisible by both `5` and `6`, neither, or just one of them. Here are some sample outputs for inputs `10`, `30`, and `23`.

```
10 is divisible by 5 or 6, but not both
30 is divisible by both 5 and 6
23 is not divisible by either 5 or 6
```

**3.13** (*An addition learning tool*) Listing 3.5, SubtractionTutor.java, randomly generates a subtraction question. Revise the program to randomly generate an addition question with two integers less than `100`.