3.1 Introduction

The program can decide which statements to execute based on a condition.

problem

If you enter a negative value for `radius` in Listing 2.2, ComputeAreaWithConsoleInput.java, the program displays an invalid result. If the radius is negative, you don't want the program to compute the area. How can you deal with this situation?

selection statements

Like all high-level programming languages, Java provides *selection statements*: statements that let you choose actions with alternative courses. You can use the following selection statement to replace lines 12–17 in Listing 2.2:

```
if (radius < 0) {
  System.out.println("Incorrect input");
}
else {
  area = radius * radius * 3.14159;
  System.out.println("Area is " + area);
}
```

Boolean expression
Boolean value

Selection statements use conditions that are Boolean expressions. A *Boolean expression* is an expression that evaluates to a *Boolean value*: `true` or `false`. We now introduce Boolean types and relational operators.

3.2 `boolean` Data Type

The `boolean` data type declares a variable with the value either `true` or `false`.

`boolean` data type
relational operators

How do you compare two values, such as whether a radius is greater than `0`, equal to `0`, or less than `0`? Java provides six *relational operators* (also known as *comparison operators*), shown in Table 3.1, which can be used to compare two values (assume radius is `5` in the table).

TABLE 3.1 Relational Operators

Java Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	less than	`radius < 0`	`false`
`<=`	≤	less than or equal to	`radius <= 0`	`false`
`>`	>	greater than	`radius > 0`	`true`
`>=`	≥	greater than or equal to	`radius >= 0`	`true`
`==`	=	equal to	`radius == 0`	`false`
`!=`	≠	not equal to	`radius != 0`	`true`

`==` vs. `=`

Caution

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

The result of the comparison is a Boolean value: `true` or `false`. For example, the following statement displays `true`:

```
double radius = 1;
System.out.println(radius > 0);
```

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. A `boolean` variable can hold one of the

two values: `true` or `false`. For example, the following statement assigns `true` to the variable `lightsOn`:

```
boolean lightsOn = true;
```

`true` and `false` are literals, just like a number such as `10`. They are treated as reserved words and cannot be used as identifiers in the program.

Boolean literals

Suppose you want to develop a program to let a first-grader practice addition. The program randomly generates two single-digit integers, `number1` and `number2`, and displays to the student a question such as "What is 1 + 7?," as shown in the sample run in Listing 3.1. After the student types the answer, the program displays a message to indicate whether it is true or false.

VideoNote
Program addition quiz

There are several ways to generate random numbers. For now, generate the first integer using `System.currentTimeMillis() % 10` and the second using `System.currentTimeMillis() / 7 % 10`. Listing 3.1 gives the program. Lines 5–6 generate two numbers, `number1` and `number2`. Line 14 obtains an answer from the user. The answer is graded in line 18 using a Boolean expression `number1 + number2 == answer`.

LISTING 3.1 AdditionQuiz.java

```
1  import java.util.Scanner;
2
3  public class AdditionQuiz {
4    public static void main(String[] args) {
5      int number1 = (int)(System.currentTimeMillis() % 10);
6      int number2 = (int)(System.currentTimeMillis() / 7 % 10);
7
8      // Create a Scanner
9      Scanner input = new Scanner(System.in);
10
11     System.out.print(
12       "What is " + number1 + " + " + number2 + "? ");
13
14     int number = input.nextInt();
15
16     System.out.println(
17       number1 + " + " + number2 + " = " + answer + " is " +
18       (number1 + number2 == answer));
19   }
20 }
```

generate number1
generate number2

show question

display result

```
What is 1 + 7? 8 Enter
1 + 7 = 8 is true
```

```
What is 4 + 8? 9 Enter
4 + 8 = 9 is false
```

line#	number1	number2	answer	output
5	4			
6		8		
14			9	
16				4 + 8 = 9 is false

Check Point

3.1 List six relational operators.

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

```
(x > 0)
(x < 0)
(x != 0)
(x >= 0)
(x != 1)
```

3.3 Can the following conversions involving casting be allowed? Write a test program to verify your answer.

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

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

3.3 `if` Statements

Key Point

An `if` statement is a construct that enables a program to specify alternative paths of execution.

The preceding program 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 to make this minor change.

why `if` statement?

Java has several types of selection statements: one-way `if` statements, two-way `if-else` statements, nested `if` statements, multi-way `if-else` statements, `switch` statements, and conditional expressions.

A one-way `if` statement executes an action if and only if the condition is `true`. The syntax for a one-way `if` statement is:

`if` statement

```
if (boolean-expression) {
  statement(s);
}
```

flowchart

The flowchart in Figure 3.1a illustrates how Java executes the syntax of an `if` statement. A *flowchart* is a diagram that describes an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Process operations are represented in these boxes, and arrows connecting them represent the flow of control. A diamond box denotes a Boolean condition and a rectangle box represents statements.

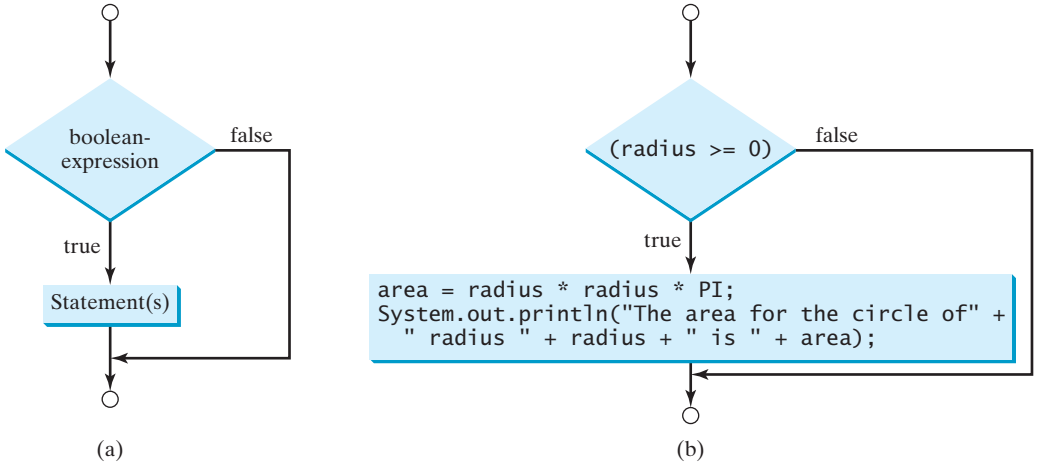


FIGURE 3.1 An `if` statement executes statements if the `boolean-expression` evaluates to `true`.

If the boolean-expression 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 flowchart of the preceding statement is shown in Figure 3.1b. 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.

The boolean-expression is enclosed in parentheses. For example, the code in (a) is wrong. It should be corrected, as shown in (b).

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

The block braces can be omitted if they enclose a single statement. For example, the following statements are equivalent.

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

Note

Omitting braces makes the code shorter, but it is prone to errors. It is a common mistake to forget the braces when you go back to modify the code that omits the braces.

Omitting braces or not

Listing 3.2 gives a program that prompts the user to enter an integer. If the number is a multiple of 5, the program displays HiFive. If the number is divisible by 2, it displays HiEven.

LISTING 3.2 SimpleIfDemo.java

```
1  import java.util.Scanner;
2
3  public class SimpleIfDemo {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      System.out.println("Enter an integer: ");
7      int number = input.nextInt();                    enter input
8
9      if (number % 5 == 0)                             check 5
10       System.out.println("HiFive");
11
12     if (number % 2 == 0)                             check even
13       System.out.println("HiEven");
14   }
15 }
```

```
Enter an integer:  4  ↵ Enter
HiEven
```

```
Enter an integer:  30  ↵ Enter
HiFive
HiEven
```

The program prompts the user to enter an integer (lines 6–7) and displays `HiFive` if it is divisible by `5` (lines 9–10) and `HiEven` if it is divisible by `2` (lines 12–13).

3.4 Write an `if` statement that assigns `1` to `x` if `y` is greater than `0`.

3.5 Write an `if` statement that increases pay by 3% if `score` is greater than `90`.

3.4 Two-Way `if-else` Statements

An `if-else` statement decides the execution path based on whether the condition is true or false.

A one-way `if` statement performs 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 a two-way `if-else` statement. The actions that a two-way `if-else` statement specifies differ based on whether the condition is `true` or `false`.

Here is the syntax for a two-way `if-else` statement:

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

The flowchart of the statement is shown in Figure 3.2.

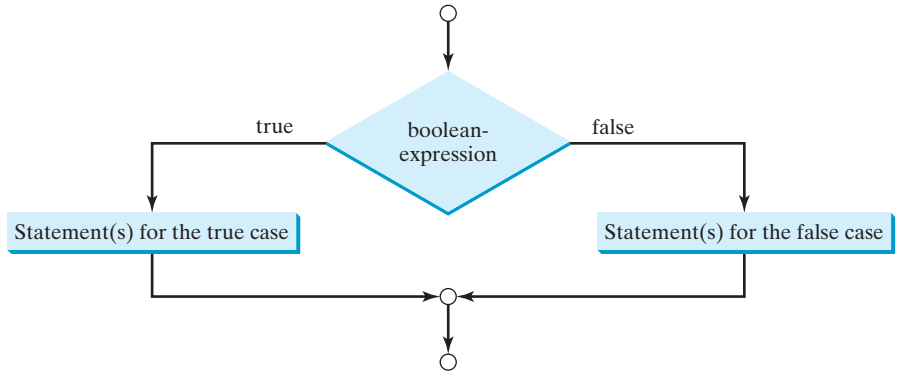


FIGURE 3.2 An `if-else` statement executes statements for the true case if the `Boolean-expression` evaluates to `true`; otherwise, statements for the false case are executed.

If the boolean-expression evaluates to true, the statement(s) for the true case are executed; otherwise, the statement(s) for the false case are executed. For example, consider the following code:

two-way 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 displayed.

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.

Here is another example of using the if-else statement. The example checks whether a number is even or odd, as follows:

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

3.6 Write an if statement that increases pay by 3% if score is greater than 90, otherwise increases pay by 1%.

3.7 What is the output of the code in (a) and (b) if number is 30? What if number is 35?

Check Point

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

System.out.println(number + " is odd.");
```

(a)

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

(b)

3.5 Nested if and Multi-Way if-else Statements

An if statement can be inside another if statement to form a nested if statement.

Key Point

nested if statement

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:

```
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 3.3a, for instance, prints a letter grade according to the score, with multiple alternatives.

```
if (score >= 90.0)
  System.out.print("A");
else
  if (score >= 80.0)
    System.out.print("B");
  else
    if (score >= 70.0)
      System.out.print("C");
    else
      if (score >= 60.0)
        System.out.print("D");
      else
        System.out.print("F");
```

(a)

Equivalent

This is better

```
if (score >= 90.0)
  System.out.print("A");
else if (score >= 80.0)
  System.out.print("B");
else if (score >= 70.0)
  System.out.print("C");
else if (score >= 60.0)
  System.out.print("D");
else
  System.out.print("F");
```

(b)

FIGURE 3.3 A preferred format for multiple alternatives is shown in (b) using a multi-way `if-else` statement.

The execution of this `if` statement proceeds as shown in Figure 3.4. The first condition `(score >= 90.0)` is tested. If it is `true`, the grade is `A`. If it is `false`, the second condition `(score >= 80.0)` is tested. If the second condition is `true`, the grade is `B`. If that condition is `false`, the third condition and the rest of the conditions (if necessary) are tested until a condition is met or all of the conditions prove to be `false`. If all of the conditions are `false`, the grade is `F`. Note that a condition is tested only when all of the conditions that come before it are `false`.

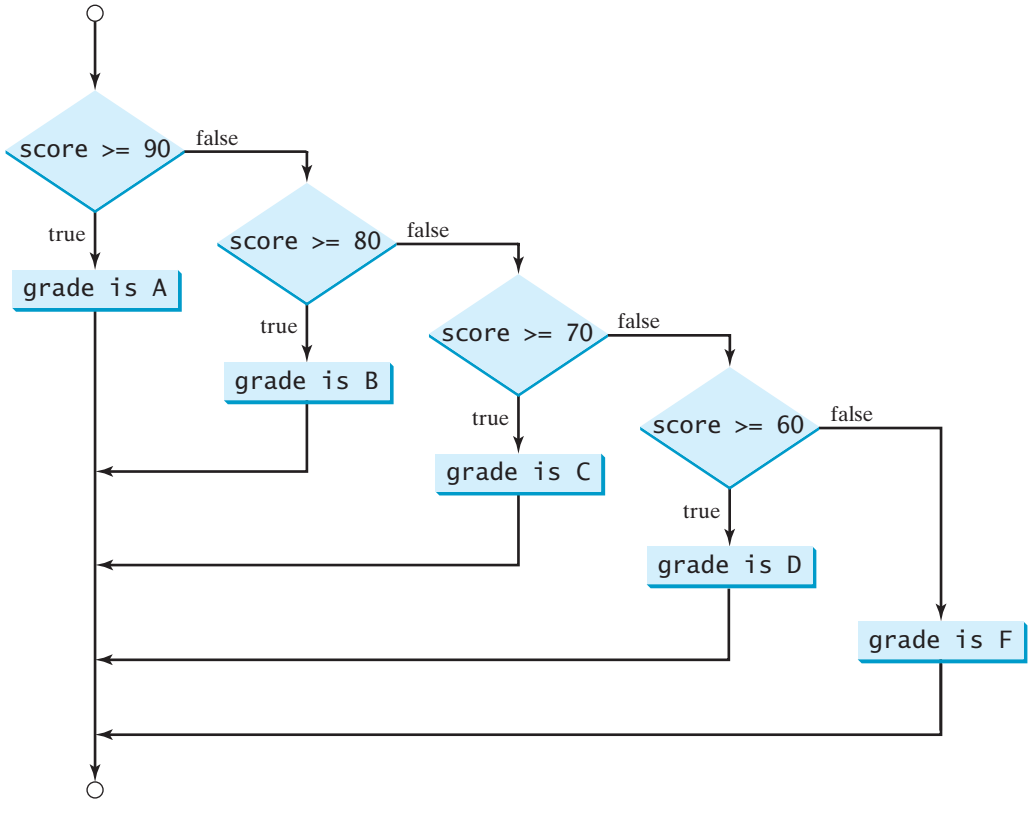


FIGURE 3.4 You can use a multi-way `if-else` statement to assign a grade.

The `if` statement in Figure 3.3a is equivalent to the `if` statement in Figure 3.3b. In fact, Figure 3.3b is the preferred coding style for multiple alternative `if` statements. This style, called *multi-way* `if-else` *statements*, avoids deep indentation and makes the program easy to read.

multi-way `if` statement

3.8 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 code.


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

3.9 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`?

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

3.10 What is wrong in the following code?

```
if (score >= 60.0)
  System.out.println("D");
else if (score >= 70.0)
  System.out.println("C");
else if (score >= 80.0)
  System.out.println("B");
else if (score >= 90.0)
  System.out.println("A");
else
  System.out.println("F");
```

3.6 Common Errors and Pitfalls

Forgetting necessary braces, ending an `if` *statement in the wrong place, mistaking* `==` *for* `=`*, and dangling* `else` *clauses are common errors in selection statements. Duplicated statements in* `if-else` *statements and testing equality of double values are common pitfalls.*

The following errors are common among new programmers.

Common Error 1: Forgetting Necessary Braces

The braces can be omitted if the block contains a single statement. However, 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. For example, the following code in (a) is wrong. It should be written with braces to group multiple statements, as shown in (b).

```
if (radius >= 0)
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
```

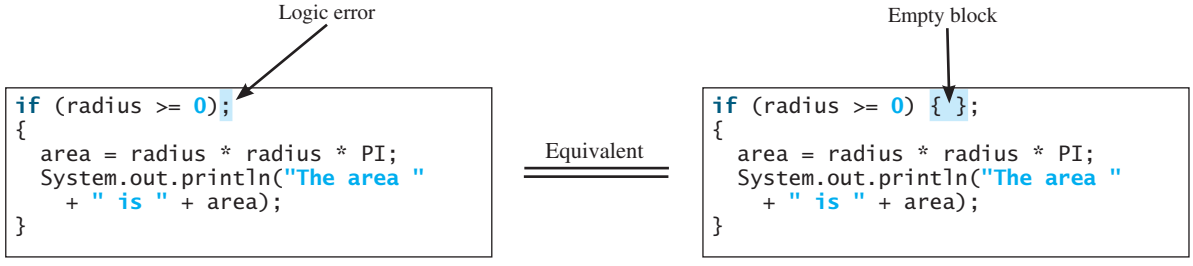
(a) Wrong

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

(b) Correct

Common Error 2: Wrong Semicolon at the `if` Line

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



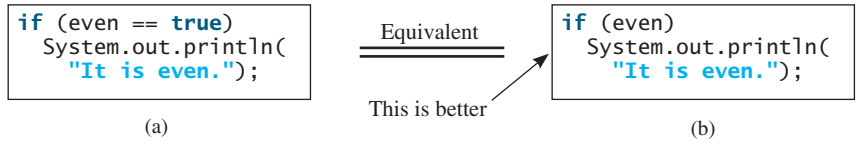
(a) (b)

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

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

Common Error 3: Redundant Testing of Boolean Values

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



(a) (b)

Instead, it is better to test the `boolean` variable directly, as shown in (b). Another good reason for doing this is to avoid errors that are difficult to detect. Using the `=` operator instead of the `==` operator to compare the 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.");
```

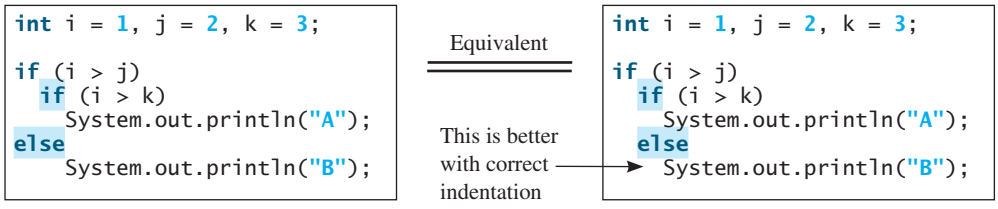
This statement does not have compile errors. It assigns `true` to `even`, so that `even` is always `true`.

Common Error 4: Dangling `else` Ambiguity

The code in (a) below has two `if` clauses and one `else` clause. Which `if` clause is matched by the `else` clause? The indentation indicates that the `else` clause matches the first `if` clause.

However, the `else` clause actually matches the second `if` clause. This situation is known as the *dangling else ambiguity*. The `else` clause always matches the most recent unmatched `if` clause in the same block. So, the statement in (a) is equivalent to the code in (b).

dangling else ambiguity



(a) (b)

Since `(i > j)` is false, nothing is displayed 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, j = 2, k = 3;

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

This statement displays `B`.

Common Error 5: Equality Test of Two Floating-Point Values

As discussed in Common Error 3 in Section 2.18, floating-point numbers have a limited precision and calculations; involving floating-point numbers can introduce round-off errors. So, equality test of two floating-point values is not reliable. For example, you expect the following code to display `true`, but surprisingly it displays `false`.

```
double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
System.out.println(x == 0.5);
```

Here, `x` is not exactly `0.5`, but is `0.5000000000000001`. You cannot reliably test equality of two floating-point values. However, you can compare whether they are close enough by testing whether the difference of the two numbers is less than some threshold. That is, two numbers x and y are very close if $|x-y| < \varepsilon$ for a very small value, ε. ε, a Greek letter pronounced epsilon, is commonly used to denote a very small value. Normally, you set ε to 10^{-14} for comparing two values of the `double` type and to 10^{-7} for comparing two values of the `float` type. For example, the following code

```
final double EPSILON = 1E-14;
double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
if (Math.abs(x - 0.5) < EPSILON)
  System.out.println(x + " is approximately 0.5");
```

will display that

```
0.5000000000000001 is approximately 0.5
```

The `Math.abs(a)` method can be used to return the absolute value of `a`.

Common Pitfall 1: Simplifying Boolean Variable Assignment

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

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

(a)

Equivalent

This is shorter

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

(b)

This is not an error, but it should be better written as shown in (b).

Common Pitfall 2: Avoiding Duplicate Code in Different Cases

Often, new programmers write the duplicate code in different cases that should be combined in one place. For example, the highlighted code in the following statement is duplicated.

```
if (inState) {
  tuition = 5000;
  System.out.println("The tuition is " + tuition);
}
else {
  tuition = 15000;
  System.out.println("The tuition is " + tuition);
}
```

This is not an error, but it should be better written as follows:

```
if (inState) {
  tuition = 5000;
}
else {
  tuition = 15000;
}
System.out.println("The tuition is " + tuition);
```

The new code removes the duplication and makes the code easy to maintain, because you only need to change in one place if the print statement is modified.

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 Rewrite the following statement using a Boolean expression:

```
if (count % 10 == 0)
  newLine = true;
else
  newLine = false;
```

3.13 Are the following statements correct? Which one is better?

```
if (age < 16)
  System.out.println
    ("Cannot get a driver's license");
if (age >= 16)
  System.out.println
    ("Can get a driver's license");
```

(a)

```
if (age < 16)
  System.out.println
    ("Cannot get a driver's license");
else
  System.out.println
    ("Can get a driver's license");
```

(b)

3.14 What is the output of the following code if `number` is `14`, `15`, or `30`?

```
if (number % 2 == 0)
  System.out.println
    (number + " is even");
if (number % 5 == 0)
  System.out.println
    (number + " is multiple of 5");
```

(a)

```
if (number % 2 == 0)
  System.out.println
    (number + " is even");
else if (number % 5 == 0)
  System.out.println
    (number + " is multiple of 5");
```

(b)

3.7 Generating Random Numbers

You can use `Math.random()` to obtain a random double value between `0.0` and `1.0`, excluding `1.0`.

Key Point

VideoNote
Program subtraction quiz

Suppose you want to develop a program for a first-grader to practice subtraction. The program randomly generates two single-digit integers, `number1` and `number2`, with `number1 >= number2`, and it displays to the student a question such as "What is 9 − 2?" After the student enters the answer, the program displays a message indicating whether it is correct.

The previous programs generate random numbers using `System.currentTimeMillis()`. A better approach is to use the `random()` method in the `Math` class. Invoking this method returns a random double value `d` such that $0.0 \leq d < 1.0$. Thus, `(int)(Math.random() * 10)` returns a random single-digit integer (i.e., a number between `0` and `9`).

`random()` method

The program can work as follows:

1. Generate two single-digit integers into `number1` and `number2`.
2. If `number1 < number2`, swap `number1` with `number2`.
3. Prompt the student to answer, `"What is number1 - number2?"`
4. Check the student's answer and display whether the answer is correct.

The complete program is shown in Listing 3.3.

LISTING 3.3 SubtractionQuiz.java

```
1  import java.util.Scanner;
2
3  public class SubtractionQuiz {
4    public static void main(String[] args) {
5      // 1. Generate two random single-digit integers
6      int number1 = (int)(Math.random() * 10);
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;
```

random number

```
12        number1 = number2;
13        number2 = temp;
14      }
15
16      // 3. Prompt the student to answer "What is number1 - number2?"
17      System.out.print
18        ("What is " + number1 + " - " + number2 + "? ");
19      Scanner input = new Scanner(System.in);
20      int answer = input.nextInt();
21
22      // 4. Grade the answer and display the result
23      if (number1 - number2 == answer)
24        System.out.println("You are correct!");
25      else {
26        System.out.println("Your answer is wrong.");
27        System.out.println(number1 + " - " + number2 +
28          " should be " + (number1 - number2));
29      }
30    }
31  }
```

get answer

check the answer

```
What is 6 - 6? 0 Enter
You are correct!
```

```
What is 9 - 2? 5 Enter
Your answer is wrong
9 - 2 is 7
```

line#	number1	number2	temp	answer	output
6	2				
7		9			
11			2		
12	9				
13		2			
20				5	
26					Your answer is wrong 9 - 2 should be 7

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` (line 13).

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

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

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

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

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

d. Write an expression that returns `0` or `1` randomly.

3.8 Case Study: Computing Body Mass Index

You can use nested `if` *statements to write a program that interprets body mass index.*

Body Mass Index (BMI) is a measure of health based on height and weight. It can be calculated by taking your weight in kilograms and dividing it by the square of your height in meters. The interpretation of BMI for people 20 years or older is as follows:

BMI	Interpretation
BMI < 18.5	Underweight
18.5 ≤ BMI < 25.0	Normal
25.0 ≤ BMI < 30.0	Overweight
30.0 ≤ BMI	Obese

Write a program that prompts the user to enter a weight in pounds and height in inches and displays the BMI. Note that one pound is `0.45359237` kilograms and one inch is `0.0254` meters. Listing 3.4 gives the program.

LISTING 3.4 ComputeAndInterpretBMI.java

```
1  import java.util.Scanner;
2
3  public class ComputeAndInterpretBMI {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      // Prompt the user to enter weight in pounds
8      System.out.print("Enter weight in pounds: ");
9      double weight = input.nextDouble();                          input weight
10
11     // Prompt the user to enter height in inches
12     System.out.print("Enter height in inches: ");
13     double height = input.nextDouble();                          input height
14
15     final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
16     final double METERS_PER_INCH = 0.0254; // Constant
17
18     // Compute BMI
19     double weightInKilograms = weight * KILOGRAMS_PER_POUND;
20     double heightInMeters = height * METERS_PER_INCH;
21     double bmi = weightInKilograms /                             compute bmi
22       (heightInMeters * heightInMeters);
23
24     // Display result
25     System.out.println("BMI is " + bmi);                         display output
26     if (bmi < 18.5)
27       System.out.println("Underweight");
28     else if (bmi < 25)
29       System.out.println("Normal");
30     else if (bmi < 30)
31       System.out.println("Overweight");
32     else
33       System.out.println("Obese");
34   }
35 }
```

```
Enter weight in pounds:  146 ↵Enter
Enter height in inches:  70 ↵Enter
BMI is 20.948603801493316
Normal
```

line#	weight	height	weightInKilograms	heightInMeters	bmi	output
9	146					
13		70				
19			66.22448602			
20				1.778		
21					20.9486	
25						BMI is 20.95
31						Normal

The constants `KILOGRAMS_PER_POUND` and `METERS_PER_INCH` are defined in lines 15–16. Using constants here makes programs easy to read.

test all cases

You should test the input that covers all possible cases for BMI to ensure that the program works for all cases.

3.9 Case Study: Computing Taxes

You can use nested `if` statements to write a program for computing taxes.

VideoNote
Use multi-way `if-else` statements

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 or qualified widow(er), married filing separately, and head of household. The tax rates vary every year. Table 3.2 shows the rates for 2009. If you are, say, single with a taxable income of $10,000, the first $8,350 is taxed at 10% and the other $1,650 is taxed at 15%, so, your total tax is $1,082.50.

TABLE 3.2 2009 U.S. Federal Personal Tax Rates

Marginal Tax Rate	*Single*	*Married Filing Jointly or Qualifying Widow(er)*	*Married Filing Separately*	*Head of Household*
10%	$0 – $8,350	$0 – $16,700	$0 – $8,350	$0 – $11,950
15%	$8,351 – $33,950	$16,701 – $67,900	$8,351 – $33,950	$11,951 – $45,500
25%	$33,951 – $82,250	$67,901 – $137,050	$33,951 – $68,525	$45,501 – $117,450
28%	$82,251 – $171,550	$137,051 – $208,850	$68,526 – $104,425	$117,451 – $190,200
33%	$171,551 – $372,950	$208,851 – $372,950	$104,426 – $186,475	$190,201 – $372,950
35%	$372,951+	$372,951+	$186,476+	$372,951+

You are to write a program to compute personal income tax. Your program should prompt the user to enter the filing status and taxable income and compute the tax. Enter `0` for single filers, `1` for married filing jointly or qualified widow(er), `2` for married filing separately, and `3` for head of household.

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 filing jointly or qualifying widow(er)
}
else if (status == 2) {
  // Compute tax for married filing 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, $8,350 is taxed at 10%, (33,950 – 8,350) at 15%, (82,250 – 33,950) at 25%, (171,550 – 82,250) at 28%, (372,950 – 171,550) at 33%, and (400,000 – 372,950) at 35%.

Listing 3.5 gives the solution for computing taxes for single filers. The complete solution is left as an exercise.

LISTING 3.5 ComputeTax.java

```
1  import java.util.Scanner;
2
3  public class ComputeTax {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter filing status
9      System.out.print("(0-single filer, 1-married jointly or " +
10       "qualifying widow(er), 2-married separately, 3-head of " +
11       "household) Enter the filing status: ");
12
13     int status = input.nextInt();                                    input status
14
15     // Prompt the user to enter taxable income
16     System.out.print("Enter the taxable income: ");
17     double income = input.nextDouble();                              input income
18
19     // Compute tax
20     double tax = 0;                                                  compute tax
21
22     if (status == 0) { // Compute tax for single filers
23       if (income <= 8350)
24         tax = income * 0.10;
25       else if (income <= 33950)
26         tax = 8350 * 0.10 + (income - 8350) * 0.15;
27       else if (income <= 82250)
28         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
29           (income - 33950) * 0.25;
30       else if (income <= 171550)
31         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
32           (82250 - 33950) * 0.25 + (income - 82250) * 0.28;
```

```
33        else if (income <= 372950)
34          tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
35            (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
36            (income - 171550) * 0.33;
37        else
38          tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
39            (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
40            (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
41      }
42      else if (status == 1) { // Left as an exercise
43        // Compute tax for married file jointly or qualifying widow(er)
44      }
45      else if (status == 2) { // Compute tax for married separately
46        // Left as an exercise
47      }
48      else if (status == 3) { // Compute tax for head of household
49        // Left as an exercise
50      }
51      else {
52        System.out.println("Error: invalid status");
53        System.exit(1);
54      }
55
56      // Display the result
57      System.out.println("Tax is " + (int)(tax * 100) / 100.0);
58    }
59  }
```

exit program

display output

```
(0-single filer, 1-married jointly or qualifying widow(er),
2-married separately, 3-head of household)
Enter the filing status:  0 Enter
Enter the taxable income:  400000 Enter
Tax is 117683.5
```

line#	status	income	tax	output
13	0			
17		400000		
20			0	
38			117683.5	
57				Tax is 117683.5

The program receives the filing status and taxable income. The multi-way **`if-else`** statements (lines 22, 42, 45, 48, 51) check the filing status and compute the tax based on the filing status.

`System.exit(status)`

`System.exit(status)` (line 53) is defined in the **`System`** class. Invoking this method terminates the program. The status **`0`** indicates that the program is terminated normally. A nonzero status code indicates abnormal termination.

An initial value of **`0`** is assigned to **`tax`** (line 20). A compile 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 compile error.

To test a program, you should provide the input that covers all cases. For this program, your input should cover all statuses (0, 1, 2, 3). For each status, test the tax for each of the six brackets. So, there are a total of 24 cases.

test all cases

Tip

For all programs, you should write a small amount of code and test it before moving on to add more code. This is called *incremental development and testing*. This approach makes testing easier, because the errors are likely in the new code you just added.

incremental development and testing

3.17 Are the following two statements equivalent?

Check Point

```
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;
```

3.10 Logical Operators

The logical operators `!`, `&&`, `||`, *and* `^` *can be used to create a compound Boolean expression.*

Key Point

Sometimes, whether a statement is executed is determined by a combination of several conditions. You can use logical operators to combine these conditions to form a compound Boolean expression. *Logical operators*, also known as *Boolean operators*, operate on Boolean values to create a new Boolean value. Table 3.3 lists the Boolean operators. Table 3.4 defines the not (`!`) operator, which negates `true` to `false` and `false` to `true`. Table 3.5 defines the and (`&&`) operator. The and (`&&`) of two Boolean operands is `true` if and only if both operands are `true`. Table 3.6 defines the or (`||`) operator. The or (`||`) of two Boolean operands is `true` if at least one of the operands is `true`. Table 3.7 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. Note that `p1 ^ p2` is the same as `p1 != p2`.

TABLE 3.3 Boolean Operators

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

TABLE 3.4 Truth Table for Operator `!`

`p`	`!p`	*Example (assume* `age = 24, weight = 140`*)*
`true`	`false`	`!(age > 18)` is `false`, because `(age > 18)` is `true`.
`false`	`true`	`!(weight == 150)` is `true`, because `(weight == 150)` is `false`.

TABLE 3.5 Truth Table for Operator &&

p_1	p_2	p_1 && p_2	*Example (assume* age = 24, weight = 140*)*
false	false	false	
false	true	false	(age > 28) && (weight <= 140) is true, because (age > 28) is false.
true	false	false	
true	true	true	(age > 18) && (weight >= 140) is true, because (age > 18) and (weight >= 140) are both true.

TABLE 3.6 Truth Table for Operator ||

p_1	p_2	p_1 || p_2	*Example (assume* age = 24, weight = 140*)*
false	false	false	(age > 34) || (weight >= 150) is false, because (age > 34) and (weight >= 150) are both false.
false	true	true	
true	false	true	(age > 18) || (weight < 140) is true, because (age > 18) is true.
true	true	true	

TABLE 3.7 Truth Table for Operator ^

p_1	p_2	p_1 ^ p_2	*Example (assume* age = 24, weight = 140*)*
false	false	false	(age > 34) ^ (weight > 140) is false, because (age > 34) and (weight > 140) are both false.
false	true	true	(age > 34) ^ (weight >= 140) is true, because (age > 34) is false but (weight >= 140) is true.
true	false	true	
true	true	false	

Listing 3.6 gives a program that checks whether a number is divisible by 2 and 3, by 2 or 3, and by 2 or 3 but not both:

LISTING 3.6 TestBooleanOperators.java

import class

input

and

```
1  import java.util.Scanner;
2
3  public class TestBooleanOperators {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Receive an input
9      System.out.print("Enter an integer: ");
10     int number = input.nextInt();
11
12     if (number % 2 == 0 && number % 3 == 0)
13       System.out.println(number + " is divisible by 2 and 3.");
14
```

```
15      if (number % 2 == 0 || number % 3 == 0)
16        System.out.println(number + " is divisible by 2 or 3.");
17
18      if (number % 2 == 0 ^ number % 3 == 0)
19        System.out.println(number +
20          " is divisible by 2 or 3, but not both.");
21    }
22  }
```

or

exclusive or

```
Enter an integer: 4 Enter
4 is divisible by 2 or 3.
4 is divisible by 2 or 3, but not both.
```

```
Enter an integer: 18 Enter
18 is divisible by 2 and 3.
18 is divisible by 2 or 3.
```

`(number % 2 == 0 && number % 3 == 0)` (line 12) checks whether the number is divisible by both `2` and `3`. `(number % 2 == 0 || number % 3 == 0)` (line 15) checks whether the number is divisible by `2` or by `3`. `(number % 2 == 0 ^ number % 3 == 0)` (line 18) checks whether the number is divisible by `2` or `3`, but not both.

Caution

In mathematics, the expression

```
1 <= numberOfDaysInAMonth <= 31
```

is correct. However, it is incorrect in Java, because `1 <= numberOfDaysInAMonth` is evaluated to a `boolean` value, which cannot be compared with `31`. Here, two operands (a `boolean` value and a numeric value) are *incompatible*. The correct expression in Java is

incompatible operands

```
(1 <= numberOfDaysInAMonth) && (numberOfDaysInAMonth <= 31)
```

Note

De Morgan's law, named after Indian-born British mathematician and logician Augustus De Morgan (1806–1871), can be used to simplify Boolean expressions. The law states:

De Morgan's law

```
!(condition1 && condition2) is the same as
  !condition1 || !condition2
!(condition1 || condition2) is the same as
  !condition1 && !condition2
```

For example,

```
! (number % 2 == 0 && number % 3 == 0)
```

can be simplified using an equivalent expression:

```
(number % 2 != 0 || number % 3 != 0)
```

As another example,

```
!(number == 2 || number == 3)
```

is better written as

```
number != 2 && number != 3
```

If one of the operands of an `&&` operator is `false`, the expression is `false`; if one of the operands of an `||` operator 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, if `p1` is `true`, evaluates `p2`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` and then, if `p1` is `false`, evaluates `p2`; if `p1` is `true`, it does not evaluate `p2`. In programming language terminology, `&&` and `||` are known as the *short-circuit* or *lazy* operators. Java also provides the unconditional AND (`&`) and OR (`|`) operators, which are covered in Supplement III.C for advanced readers.

short-circuit operator
lazy operator

3.18 Assuming 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)
```

3.19 (a) Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100`. (b) 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.

3.20 (a) Write a Boolean expression for $|x - 5| < 4.5$. (b) Write a Boolean expression for $|x - 5| > 4.5$.

3.21 Assume that `x` and `y` are `int` type. Which of the following are legal Java expressions?

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

3.22 Are the following two expressions the same?

```
a. x % 2 == 0 && x % 3 == 0
b. x % 6 == 0
```

3.23 What is the value of the expression `x >= 50 && x <= 100` if `x` is `45`, `67`, or `101`?

3.24 Suppose, when you run the following program, you enter the input `2 3 6` from the console. What is the output?

```
public class Test {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    double x = input.nextDouble();
    double y = input.nextDouble();
    double z = input.nextDouble();

    System.out.println("(x < y && y < z) is " + (x < y && y < z));
    System.out.println("(x < y || y < z) is " + (x < y || y < z));
    System.out.println("!(x < y) is " + !(x < y));
    System.out.println("(x + y < z) is " + (x + y < z));
    System.out.println("(x + y > z) is " + (x + y > z));
  }
}
```

3.25 Write a Boolean expression that evaluates to `true` if `age` is greater than `13` and less than `18`.

3.26 Write a Boolean expression that evaluates to `true` if `weight` is greater than `50` pounds or height is greater than `60` inches.

3.27 Write a Boolean expression that evaluates to `true` if `weight` is greater than `50` pounds and height is greater than `60` inches.

3.28 Write a Boolean expression that evaluates to `true` if either `weight` is greater than `50` pounds or height is greater than `60` inches, but not both.

3.11 Case Study: Determining 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.

You can use the following Boolean expressions to check whether a year is a leap year:

```
// A leap year is divisible by 4
boolean isLeapYear = (year % 4 == 0);

// A leap year is divisible by 4 but not by 100
isLeapYear = isLeapYear && (year % 100 != 0);

// A leap year is divisible by 4 but not by 100 or divisible by 400
isLeapYear = isLeapYear || (year % 400 == 0);
```

Or you can combine all these expressions into one like this:

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

Listing 3.7 gives the program that lets the user enter a year and checks whether it is a leap year.

LISTING 3.7 LeapYear.java

```
1  import java.util.Scanner;
2
3  public class LeapYear {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7      System.out.print("Enter a year: ");
8      int year = input.nextInt();                                    input
9
10     // Check if the year is a leap year
11     boolean isLeapYear =                                           leap year?
12       (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
13
14     // Display the result
15     System.out.println(year + " is a leap year? " + isLeapYear);   display result
16   }
17 }
```

```
Enter a year:  2008 Enter
2008 is a leap year? true
```

```
Enter a year:  1900 Enter
1900 is a leap year? false
```

```
Enter a year:  2002 ↵ Enter
2002 is a leap year? false
```

3.12 Case Study: Lottery

The lottery program involves generating random numbers, comparing digits, and using Boolean operators.

Suppose you want to develop a program to play lottery. The program randomly generates a lottery of a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rules:

1. If the user input matches the lottery number in the exact order, the award is $10,000.
2. If all digits in the user input match all digits in the lottery number, the award is $3,000.
3. If one digit in the user input matches a digit in the lottery number, the award is $1,000.

Note that the digits of a two-digit number may be 0. If a number is less than 10, we assume the number is preceded by a 0 to form a two-digit number. For example, number 8 is treated as 08 and number 0 is treated as 00 in the program. Listing 3.8 gives the complete program.

LISTING 3.8 Lottery.java

```
1  import java.util.Scanner;
2
3  public class Lottery {
4    public static void main(String[] args) {
5      // Generate a lottery number
6      int lottery = (int)(Math.random() * 100);
7
8      // Prompt the user to enter a guess
9      Scanner input = new Scanner(System.in);
10     System.out.print("Enter your lottery pick (two digits): ");
11     int guess = input.nextInt();
12
13     // Get digits from lottery
14     int lotteryDigit1 = lottery / 10;
15     int lotteryDigit2 = lottery % 10;
16
17     // Get digits from guess
18     int guessDigit1 = guess / 10;
19     int guessDigit2 = guess % 10;
20
21     System.out.println("The lottery number is " + lottery);
22
23     // Check the guess
24     if (guess == lottery)
25       System.out.println("Exact match: you win $10,000");
26     else if (guessDigit2 == lotteryDigit1
27           && guessDigit1 == lotteryDigit2)
28       System.out.println("Match all digits: you win $3,000");
29     else if (guessDigit1 == lotteryDigit1
30           || guessDigit1 == lotteryDigit2
31           || guessDigit2 == lotteryDigit1
32           || guessDigit2 == lotteryDigit2)
33       System.out.println("Match one digit: you win $1,000");
```

generate a lottery number

enter a guess

exact match?

match all digits?

match one digit?

```
34      else
35        System.out.println("Sorry, no match");
36    }
37  }
```

```
Enter your lottery pick (two digits): 15 ↵ Enter
The lottery number is 15
Exact match: you win $10,000
```

```
Enter your lottery pick (two digits): 45 ↵ Enter
The lottery number is 54
Match all digits: you win $3,000
```

```
Enter your lottery pick:  23 ↵ Enter
The lottery number is 34
Match one digit: you win $1,000
```

```
Enter your lottery pick: 23 ↵ Enter
The lottery number is 14
Sorry: no match
```

variable \ line#	6	11	14	15	18	19	33
lottery	34						
guess		23					
lotteryDigit1			3				
lotteryDigit2				4			
guessDigit1					2		
guessDigit2						3	
Output							Match one digit: you win $1,000

The program generates a lottery using the `random()` method (line 6) and prompts the user to enter a guess (line 11). Note that `guess % 10` obtains the last digit from `guess` and `guess / 10` obtains the first digit from `guess`, since `guess` is a two-digit number (lines 18–19).

The program checks the guess against the lottery number in this order:

1. First, check whether the guess matches the lottery exactly (line 24).
2. If not, check whether the reversal of the guess matches the lottery (lines 26–27).
3. If not, check whether one digit is in the lottery (lines 29–32).
4. If not, nothing matches and display `"Sorry, no match"` (lines 34–35).

3.13 `switch` Statements

A `switch` statement executes statements based on the value of a variable or an expression.

The `if` statement in Listing 3.5, ComputeTax.java, 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 simplify coding for multiple conditions. You can write the following `switch` statement to replace the nested `if` statement in Listing 3.5:

```
switch (status) {
  case 0:  compute tax for single filers;
           break;
  case 1:  compute tax for married jointly or qualifying widow(er);
           break;
  case 2:  compute tax for married filing separately;
           break;
  case 3:  compute tax for head of household;
           break;
  default: System.out.println("Error: invalid status");
           System.exit(1);
}
```

The flowchart of the preceding `switch` statement is shown in Figure 3.5.

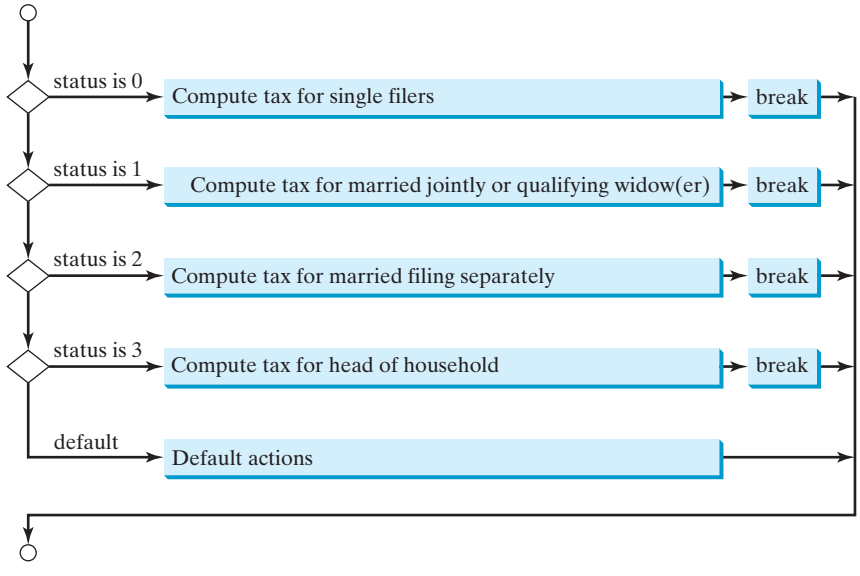


FIGURE 3.5 The `switch` statement checks all cases and executes the statements in the matched case.

`switch` statement

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 (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`, `int`, or `String` type and must always be enclosed in parentheses. (The `char` and `String` types will be introduced in the next chapter.)
- 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, 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 `default` case, which is optional, can be used to perform actions when none of the specified cases matches the `switch-expression`.
- The keyword `break` is optional. The `break` statement immediately ends the `switch` statement.

Caution

without `break`

fall-through behavior

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 is referred to as *fall-through* behavior. For example, the following code displays `Weekdays` for day of `1` to `5` and `Weekends` for day `0` and `6`.

```
switch (day) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5: System.out.println("Weekday"); break;
  case 0:
  case 6: System.out.println("Weekend");
}
```

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.

Now let us write a program to find out the Chinese Zodiac sign for a given year. The Chinese Zodiac is based on a twelve-year cycle, with each year represented by an animal—monkey, rooster, dog, pig, rat, ox, tiger, rabbit, dragon, snake, horse, or sheep—in this cycle, as shown in Figure 3.6.

Note that `year % 12` determines the Zodiac sign. 1900 is the year of the rat because `1900 % 12` is `4`. Listing 3.9 gives a program that prompts the user to enter a year and displays the animal for the year.

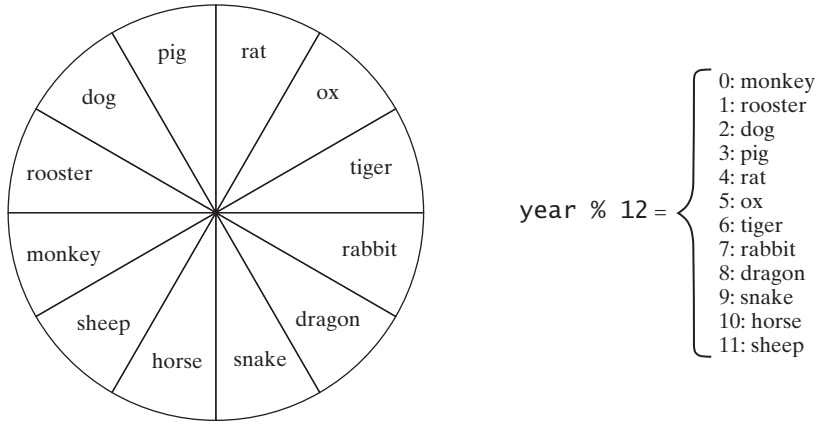


FIGURE 3.6 The Chinese Zodiac is based on a twelve-year cycle.

LISTING 3.9 ChineseZodiac.java

enter year

determine Zodiac sign

```
1  import java.util.Scanner;
2
3  public class ChineseZodiac {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      System.out.print("Enter a year: ");
8      int year = input.nextInt();
9
10     switch (year % 12) {
11       case 0: System.out.println("monkey"); break;
12       case 1: System.out.println("rooster"); break;
13       case 2: System.out.println("dog"); break;
14       case 3: System.out.println("pig"); break;
15       case 4: System.out.println("rat"); break;
16       case 5: System.out.println("ox"); break;
17       case 6: System.out.println("tiger"); break;
18       case 7: System.out.println("rabbit"); break;
19       case 8: System.out.println("dragon"); break;
20       case 9: System.out.println("snake"); break;
21       case 10: System.out.println("horse"); break;
22       case 11: System.out.println("sheep");
23     }
24   }
25 }
```

```
Enter a year:  1963 ↵ Enter
rabbit
```

```
Enter a year:  1877 ↵ Enter
ox
```

3.29 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?

3.30 What is `y` after the following `switch` statement is executed? Rewrite the code using an `if-else` statement.

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

3.31 What is `x` after the following `if-else` statement is executed? Use a `switch` statement to rewrite it and draw the flowchart for the new `switch` statement.

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

3.32 Write a `switch` statement that displays Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, if `day` is `0`, `1`, `2`, `3`, `4`, `5`, `6`, accordingly.

3.14 Conditional Expressions

A conditional expression evaluates an expression based on a condition.

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 the following example, you can use a conditional expression to achieve the same result.

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

conditional expression

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

```
boolean-expression ? expression1 : expression2;
```

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

Suppose you want to assign the larger number of 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");
```

As you can see from these examples, conditional expressions enable you to write short and concise code.

Note

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

conditional operator
ternary operator

3.33 Suppose that, when you run the following program, you enter the input `2 3 6` from the console. What is the output?

```
public class Test {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    double x = input.nextDouble();
    double y = input.nextDouble();
    double z = input.nextDouble();

    System.out.println((x < y && y < z) ? "sorted" : "not sorted");
  }
}
```

3.34 Rewrite the following `if` statements using the conditional operator.

```
if (ages >= 16)
  ticketPrice = 20;
else
  ticketPrice = 10;
```

3.35 Rewrite the following conditional expressions using `if-else` statements.

a. `score = (x > 10) ? 3 * scale : 4 * scale;`
b. `tax = (income > 10000) ? income * 0.2 : income * 0.17 + 1000;`
c. `System.out.println((number % 3 == 0) ? i : j);`

3.36 Write conditional expression that returns `-1` or `1` randomly.

3.15 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated.

Section 2.11 introduced operator precedence involving arithmetic operators. This section discusses operator precedence in more detail. Suppose that you have this expression:

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

What is its value? What is the execution order of the operators?

The expression within parentheses is evaluated first. (Parentheses can be nested, in which case the expression within 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 3.8, which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. The logical operators have lower precedence than the relational operators and the relational operators have lower precedence than the arithmetic operators. 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.)

operator precedence

TABLE 3.8 Operator Precedence Chart

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

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   is equivalent to   ((a - b) + c) - d
```

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

```
a = b += c = 5   is equivalent to   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.

Note

Java has its own way to evaluate an expression internally. The result of a Java evaluation is the same as that of its corresponding arithmetic evaluation. Advanced readers may refer to Supplement III.B for more discussions on how an expression is evaluated in Java *behind the scenes*.

behind the scenes

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


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

3.38 True or false? All the binary operators except `=` are left associative.

3.39 Evaluate the following expressions:

```
2 * 2 - 3 > 2 && 4 - 2 > 5
2 * 2 - 3 > 2 || 4 - 2 > 5
```

3.40 Is `(x > 0 && x < 10)` the same as `((x > 0) && (x < 10))`? Is `(x > 0 || x < 10)` the same as `((x > 0) || (x < 10))`? Is `(x > 0 || x < 10 && y < 0)` the same as `(x > 0 || (x < 10 && y < 0))`?

3.16 Debugging

Debugging is the process of finding and fixing errors in a program.

As mentioned in Section 1.10.1, syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are there. Runtime errors are not difficult to find either, because the Java interpreter displays them on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

bugs
debugging
hand-traces

Logic errors are called *bugs*. The process of finding and correcting errors is called *debugging*. A common approach to debugging is to use a combination of methods to help pinpoint the part of the program where the bug is located. You can *hand-trace* the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. These approaches might work for debugging a short, simple program, but for a large, complex program, the most effective approach is to use a debugger utility.

JDK includes a command-line debugger, jdb, which is invoked with a class name. jdb is itself a Java program, running its own copy of Java interpreter. All the Java IDE tools, such as Eclipse and NetBeans, include integrated debuggers. The debugger utilities let you follow the execution of a program. They vary from one system to another, but they all support most of the following helpful features.

- **Executing a single statement at a time:** The debugger allows you to execute one statement at a time so that you can see the effect of each statement.
- **Tracing into or stepping over a method:** If a method is being executed, you can ask the debugger to enter the method and execute one statement at a time in the method, or you can ask it to step over the entire method. You should step over the entire method if you know that the method works. For example, always step over system-supplied methods, such as `System.out.println`.
- **Setting breakpoints:** You can also set a breakpoint at a specific statement. Your program pauses when it reaches a breakpoint. You can set as many breakpoints as you want. Breakpoints are particularly useful when you know where your programming error starts. You can set a breakpoint at that statement and have the program execute until it reaches the breakpoint.
- **Displaying variables:** The debugger lets you select several variables and display their values. As you trace through a program, the content of a variable is continuously updated.
- **Displaying call stacks:** The debugger lets you trace all of the method calls. This feature is helpful when you need to see a large picture of the program-execution flow.
- **Modifying variables:** Some debuggers enable you to modify the value of a variable when debugging. This is convenient when you want to test a program with different samples but do not want to leave the debugger.

debugging in IDE

Tip

If you use an IDE such as Eclipse or NetBeans, please refer to *Learning Java Effectively with Eclipse/NetBeans* in Supplements II.C and II.E on the Companion Website. The supplement shows you how to use a debugger to trace programs and how debugging can help in learning Java effectively.