3.1 Introduction

- In this chapter, you will learn various selection and loop control statements.
- Java provides selection statements that let you choose actions with two or more alternative courses.
- Java provides a powerful control structure called a loop, which controls how many times an operation or a sequence of operation is performed in succession.

3.2 Selection Statements

- Java has several types of selection statements:
  - if Statements, if ... else statements, nested if statements
  - switch Statements
  - Conditional Expressions

3.2.1 Simple if Statements

```java
if (booleanExpression) {
    statement(s);
}    // execution flow chart is shown in Figure (A)
```

Example

```java
if (radius >= 0) {
    area = radius * radius * PI;
    System.out.println("The area for the circle of radius " +
    radius + " is " + area);
} // if the Boolean expression evaluates to T, the statements in the block are executed as shown in figure (B)
```

FIGURE 3.1 An if statement executes statements if the Boolean Expression evaluates as true
- **Note:** The Boolean expression is enclosed in parentheses for all forms of the if statement. Thus, the outer parentheses in the previous if statements are required.

- **Caution:**
  - Adding a semicolon at the end of an if clause is a common mistake.
  - This mistake is hard to find, because it is not a compilation error or a runtime error, it is a **logic error**.
  - This error often occurs when you use the next-line block style.

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

3.2.2 The if...else Statements

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

**FIGURE 3.2** An if … else executes statements for the true case if the Boolean expression evaluations are true; otherwise, statements for the false case are executed.
if...else Example

```java
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"); // braces may be omitted
}
```

- If radius >= 0 is true, area is computed and displayed; if it is false, the message “Negative input” is printed.
- Using the if ... else statement, you can rewrite the following code for determining whether a number is even or odd, as follows:

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

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

3.2.3 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.
- There is no limit to the depth of the nesting.

```java
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) is nested inside the if (i > k)
```
The nested if statement can be used to implement multiple alternatives.

```java
if (score >= 90)
    grade = 'A';
else
    if (score >= 80)
        grade = 'B';
    else
        if (score >= 70)
            grade = 'C';
        else
            if (score >= 60)
                grade = 'D';
            else
                grade = 'F';
```

The preceding if statement is equivalent to the following preferred format because it is easier to read:

```java
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'F';
```

Note:
- The else clause matches the most recent unmatched if clause in the same block. For example, the following statement:

```java
int i = 1; int j = 2; int k = 3;
if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

is equivalent to:

```java
int i = 1; int j = 2; int k = 3;
if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

- Nothing is printed from the preceding statement because the compiler ignores indentation. 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) {
    if (i > k)
        System.out.println("A");
} else
    System.out.println("B");

This statement prints 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 this:

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

Instead, it is **better** to use the Boolean variable directly, as follows:

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

- What’s wrong with the following?

    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.

- This problem is taken from example 3.1 (Page 86). Write a program that prompts the user to enter the filing status and taxable income and computes the tax for the year 2002.

```java
import javax.swing.JOptionPane;
public class ComputeTaxWithSelectionStatement {
    public static void main(String[] args) {
        // Prompt the user to enter filing status
        String statusString = JOptionPane.showInputDialog(null,
            "Enter the filing status:
            (0-single filer, 1-married jointly,
            2-married separately, 3-head of household)",
            "Example 3.1 Input", JOptionPane.QUESTION_MESSAGE);
        int status = Integer.parseInt(statusString);
        // Prompt the user to enter taxable income
        String incomeString = JOptionPane.showInputDialog(null,
            "Enter the taxable income:",
            "Example 3.1 Input", JOptionPane.QUESTION_MESSAGE);
        double income = Double.parseDouble(incomeString);
        // Compute tax
        double tax = 0;
        if (status == 0) {
            // Compute tax for single filers
            if (income <= 6000)
                tax = income * 0.1;
        }
    }
}
```
tax = income * 0.10;
else if (income <= 27950)
tax = 6000 * 0.10 + (income - 6000) * 0.15;
else if (income <= 67700)
tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
    (income - 27950) * 0.27;
else if (income <= 141250)
tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
    (67700 - 27950) * 0.27 + (income - 67700) * 0.30;
else if (income <= 307050)
tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
    (67700 - 27950) * 0.27 + (141250 - 67700) * 0.30 +
    (income - 141250) * 0.35;
else
tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
    (67700 - 27950) * 0.27 + (141250 - 67700) * 0.30 +
    (307050 - 141250) * 0.35 + (income - 307050) * 0.386;
else if (status == 1) { // Compute tax for married file jointly, 
    // married separately, and head of household Left as exercise
}
else if (status == 2) {} 
else if (status == 3) {} 
else {
    System.out.println("Error: invalid status");
    System.exit(0);
}
// Display the result
JOptionPane.showMessageDialog(null, "Tax is " +
    (int)(tax * 100) / 100.0,
    "Example 3.1 Output", JOptionPane.INFORMATION_MESSAGE);
}

3.2.4 The switch Statements

- One can write a switch statement to replace a nested if statement. For example,

```java
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);
} // checks if status matches the values 0, 1, 2, or 3 respectively.
```
The switch Statement 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. 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 cases statements are checked in sequential order, but the order of the cases (including the default case) does not matter. However, it is a good programming style to follow the logical sequence of the cases and place the default case at the end.

Caution

- Do not forget to use a break statement when one is needed. For example, the following code prints character a tree times if ch is ‘a’:

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

- Conditional expressions are in different style, which no explicit if in the statement. The syntax is shown below:

  BooleanExpression ? expression1 : expression2;

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

- For example:

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

  is equivalent to

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

- For example:

  ```java
  if (num % 2 == 0)
      System.out.println(num + “is even”);
  else
      System.out.println(num + “is odd”);
  ```

  is equivalent to

  ```java
  System.out.println((num % 2 == 0)? num + “is even” : num + “is odd”);
  ```

- For example:

  ```java
  Max = (num1 > num2) ? num1 : num2;
  ```

Note

- The symbols ? and : appear together in a **conditional expression**. They form a condition operator. The operator is called a **ternary** operator because it uses three operands.
3.3 Loop Statements

- **Loops** are structures that control **repeated** executions of a block of statements.
- Part of the loop that contains the statements to be repeated is called the **loop body**.
- A one-time execution of a loop body is referred to as an **iteration of the loop**.
- Each loop contains a **loop-continuation-condition**, a Boolean expression that controls the execution of the body.
- After each iteration, the **loop-continuation-condition** is **reevaluated**. If the condition is **true**, the execution of the loop body is repeated. If the condition is **false**, the loop terminates.

3.3.1 The **While Loop**

- The syntax for the while loop is as follows:

  ```java
  while (loop-continuation-condition) {
    // loop-body
    Statement(s);
  }
  ```

  - The braces enclosing a while loop or any other loop can be omitted only if the loop body contains one or no statement. The **while** loop flowchart is in Figure (a).
  - The **loop-continuation-condition**, a Boolean expression, must appear inside the parentheses. It is always evaluated **before** the loop body is executed.
  - If its evaluation is **true**, the loop body is executed; if its evaluation is false, the entire loop terminates, and the program control turns to the statement that follows the while loop. For example, the following while loop prints **Welcome to Java!** 100 times.

    ```java
    int count = 0;
    while (count < 100) {
      System.out.println("Welcome to Java!");
      count++;
    }
    ```

![Flowchart](image)

**FIGURE 3.6** The while loop repeatedly executes the statements in the loop body when the loop-continuation-condition evaluates as true.
Caution

- Make sure that the loop-continuation-condition eventually becomes false so that the program will terminate.
- A common programming error involves infinite loops.

EXAMPLE 3.2 Using while Loops (Page 92)

```java
// TestWhile.java: Test the while loop
import javax.swing.JOptionPane;

public class TestWhile {
    /** Main method */
    public static void main(String[] args) {
        int data;
        int sum = 0;

        // Read an initial data
        String dataString = JOptionPane.showInputDialog(null,
                "Enter an int value, \nthe program exits if the input is 0",
                "Example 3.2 Input", JOptionPane.QUESTION_MESSAGE);

        data = Integer.parseInt(dataString);

        // Keep reading data until the input is 0
        while (data != 0) {
            sum += data;

            // Read the next data
            dataString = JOptionPane.showInputDialog(null,
                    "Enter an int value, \nthe program exits if the input is 0",
                    "Example 3.2 Input", JOptionPane.QUESTION_MESSAGE);

            data = Integer.parseInt(dataString);

        }

        JOptionPane.showMessageDialog(null, "The sum is " + sum,
                "Example 3.2 Output", JOptionPane.INFORMATION_MESSAGE);

        System.exit(0);
    }
}
```

- If data is not 0, it is added to the sum and the next input data are read. If data is 0, the loop body is not executed and the while loop terminates.
- If the first input read is 0, the loop body never executes, and the resulting sum is 0.
- The do-while loop executes the loop body first, and then checks the loop-continuation condition to determine whether to continue or terminate the loop.

Caution

- Don’t use floating-point values for equality checking in a loop control. Since floating-point values are approximations, using them could result in imprecise counter values and inaccurate results. This example uses int value for data. If a
floating-point type value is used for data, (data != 0) may be true even though data is 0.

```java
// data should be zero
double data = Math.pow(Math.sqrt(2), 2) - 2;
if (data == 0)
    System.out.println("data is zero");
else
    System.out.println("data is not zero");
```

- Like pow, sqrt is a method in the Math class for computing the square root of a number.

### 3.3.2 The do-while Loop

- The do-while is a variation of the while-loop. Its syntax is shown below.

```java
do {
    // Loop body
    Statement(s);
} while (continue-condition);  // Do forget ";"
```

- The loop body is executed first. Then the loop-continuation-condition is evaluated. If the evaluation is true, the loop body is executed again; if it is false, the do-while loop terminates.
- The major difference between a while loop and a do-while loop is the order in which the loop-continuation-condition is evaluated and the loop body executed.
- The while loop and the do-while loop have equal expressive power.
- Sometimes one is a more convenient choice than the other.
- **Tip:** Use the do-while loop if you have statements inside the loop that must be executed at least once.
For example (Page 94), you can rewrite the TestWhile program shown previously as follows:

```java
// TestDo.java: Test the do-while loop
import javax.swing.JOptionPane;

public class TestDoWhile {
    /** Main method */
    public static void main(String[] args) {
        int data;
        int sum = 0;

        // Keep reading data until the input is 0
        do {
            // Read the next data
            String dataString = JOptionPane.showInputDialog(null,
                "Enter an int value, \nthe program exits if the input is 0",
                "TestDo", JOptionPane.QUESTION_MESSAGE);

            data = Integer.parseInt(dataString);
            sum += data;
        } while (data != 0);

        JOptionPane.showMessageDialog(null, "The sum is " + sum,
            "TestDo", JOptionPane.INFORMATION_MESSAGE);

        System.exit(0);
    }
}
```

### 3.3.3 The for loop

- The syntax of a for loop is as shown below.

```java
for (initial-action; loop-continuation-condition;
    action-after-each-iteration) {
    //loop body;
    Statement(s);
}
```

- The for loop statement starts with the keyword for, followed by a pair of parentheses enclosing initial-action, loop-continuation-condition, and action-after-each-iteration, and the loop body, enclosed inside braces.
- initial-action, loop-continuation-condition, and action-after-each-iteration are separated by semicolons;
- A for loop generally uses a variable to control how many times the loop body is executed and when the loop terminates.
- This variable is referred to as a control variable. The initial-action often initializes a control variable, the action-after-each-iteration usually increments or decrements the control variable, and the loop-continuation-condition tests whether the control variable has reached a termination value.
Example: The following while loop prints Welcome to Java! 100 times.

```java
int i = 0;
while (i < 100) {
    System.out.println("Welcome to Java!");
    i++;
}
```

Example: The following for loop prints Welcome to Java! 100 times.

```java
int i;
for (i = 0; i < 100; i++) {
    System.out.println("Welcome to Java!");
}
```

- The for loop initializes `i` to 0, then repeatedly executes the `println` and evaluates `i++` if `i` is less than 100.
- The initial-action, `i = 0`, initializes the control variable, `i`.
- The loop-continuation-condition, `i < 100`, is a Boolean expression.
- The expression is evaluated at the beginning of each iteration.
- If the condition is true, execute the loop body. If it is false, the loop terminates and the program control turns to the line following the loop.
- The action-after-each-iteration, `i++`, is a statement that adjusts the control variable.
- This statement is executed after each iteration. It increments the control variable.
- Eventually, the value of the control variable forces the loop-continuation-condition to become false.
The loop control variable can be declared and initialized in the `for` loop as follows:

```java
for (int i = 0; i < 100; i++) {
    System.out.println("Welcome to Java");
}
```

**Note**

- The initial-action in a `for` loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions.

```java
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
    // Do something
}
```

- The action-after-each-iteration in a `for` loop can be a list of zero or more comma-separated statements. The following is correct but not a good example, because it makes the code hard to read.

```java
for (int i = 1; i < 100; System.out.println(i), i++);
```

**Note**

- If the loop-continuation-condition in a `for` loop is omitted, it is implicitly `true`. Thus the statement given below in (A), which is an infinite loop, is correct. Nevertheless, I recommend that you use the equivalent loop in (B) to avoid confusion:

<table>
<thead>
<tr>
<th>for ( ; ; ) { // Do something }</th>
<th>Equivalent while (true) { // Do something }</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
</tr>
</tbody>
</table>

**Page 97, EXAMPLE 3.3 Using for Loops**

- Write a program that sums a series that starts with 0.01 and ends with 1.0. The numbers in the series will increment by 0.01, as follows 0.01 + 0.02 + 0.03 and so on.

```java
// TestSum.java: Compute sum = 0.01 + 0.02 + ... + 1;
import javax.swing.JOptionPane;
public class TestSum {
    /** Main method */
    public static void main(String[] args) {
        // Initialize sum
        float sum = 0;

        // Keep adding 0.01 to sum
        for (float i = 0.01f; i <= 1.0f; i = i + 0.01f)
            sum += i;
    }
}
```
// Display result
JOptionPane.showMessageDialog(null, "The summation is " + sum,
    "Example 3.3 Output", JOptionPane.INFORMATION_MESSAGE);

System.exit(0);
}
}

- The for loop repeatedly adds the control variable i to the sum. This variable, which begins with 0.01, is incremented by 0.01 after each iteration. The loop terminates when i exceeds 1.0.
- The exact sum should be **50.50**, but the answer is **50.499985**. The result is not precise because computers use a fixed number of bits to represent floating-point numbers, and thus cannot represent some floating-point number exactly.

### 3.3.4 Nested Loops
- Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and all the required iterations are performed.

- **Page 99, EXAMPLE 3.4 Displaying the Multiplication Table**
- **Problem:** Write a program that uses nested for loops to print a multiplication table.

```java
// TestMulTable.java: Display a multiplication table
import javax.swing.JOptionPane;

public class TestMulTable {
    /** Main method */
    public static void main(String[] args) {
        // Display the table heading
        String output = " Multiplication Table
                        -----------------------------
                        | 1 2 3 4 5 6 7 8 9
                        | 1 1 2 3 4 5 6 7 8 9
                        | 2 2 4 6 8 10 12 14 16 18
                        | 3 3 6 9 12 15 18 21 24 27
                        | 4 4 8 12 16 20 24 28 32 36
                        | 5 5 10 15 20 25 30 35 40 45
                        | 6 6 12 18 24 30 36 42 48 54
                        | 7 7 14 21 28 35 42 49 56 63
                        | 8 8 16 24 32 40 48 56 64 72
                        | 9 9 18 27 36 45 54 63 72 81
                        //
                        OK
```

// TestMulTable.java: Display a multiplication table
import javax.swing.JOptionPane;

public class TestMulTable {
    /** Main method */
    public static void main(String[] args) {
        // Display the table heading
        String output = " Multiplication Table\n";
```
output += "-------------------------------------------------\n";

// Display the number title
output += "   | ";
for (int j = 1; j <= 9; j++)
    output += "    " + j;

output += "\n";
// Print table body
for (int i = 1; i <= 9; i++) {
    output += i + " | ";
    for (int j = 1; j <= 9; j++) {
        // Display the product and align properly
        if (i * j < 10)
            output += "    " + i * j;
        else
            output += "  " + i * j;
    }
    output += "\n";
}

// Display result
JOptionPane.showMessageDialog(null, output,
    "Example 3.4 Output", JOptionPane.INFORMATION_MESSAGE);
System.exit(0);
}

- The program displays a title on the first line and dashes on the second line. The first for loop displays the numbers 1 - 9 on the third line.
- The next loop is a nested for loop with the loop with the control variable i in the outer loop and j in the inner loop.
- For each i, the product i * j is displayed on a line in the inner loop, with j being 1, 2, 3, ..., 9.
- The if statement in the inner loop is used so that the product will be aligned properly.
- If the product is a single digit, it is displayed with an extra space before it.

3.5 Which Loop to Use?

- The three forms of loop statements, while, do, and for, are expressively equivalent; that is, you can write a loop in any of these three forms.
- For example, a while loop in (A) in the following figure can always be converted into the following for loop in (B):

| while (loop-continuation-condition) {   | Equivalent for (; loop-continuation-condition; ) { |
|   // Loop body                       |   // Loop body                              |
| }                                     | }                                          |

- A for loop in (A) in the following figure can generally be converted into the following while loop in (B) except in certain special cases.
Recommendations

- The author recommends that you use the one that is most intuitive and comfortable for you.
- In general, a for loop may be used if the number of repetitions is known, as, for example, when you need to print a message 100 times.
- A while loop may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0.
- A do-while loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.

Caution

- Adding a semicolon at the end of the for clause before the loop body is a common mistake, as shown below:

```java
for (int i = 0; i < 10; i++); // Logic Error (';')
{
    System.out.println("i is " + i);
}
```

- Similarly, the following loop is also wrong:

```java
int i=0;
while (i<10);       // Logic Error (';')
{
    System.out.println("i is " + i);
    i++;
}
```

- In the case of the do loop, the following semicolon is needed to end the loop.

```java
int i=0;
do {
    System.out.println("i is " + i);
    i++;
} while (i<10);    // Correct, The semicolon is needed
```
3.6 Using the Keywords *break and continue*

- The **break** control **immediately ends the innermost loop** that contains it. It is generally used with an *if* statement.
- The **continue** control **only ends the current iteration**. Program control goes to the end of the loop body. This keyword is generally used with an *if* statement.
- The **break** statement forces its containing loop to exit.

- The **continue** statement forces the current iteration of the loop to end.
Page 102, Example 3.5 Demonstrating a break Statement

// TestBreak.java: Test the break keyword in the loop
public class TestBreak {
    /** Main method */
    public static void main(String[] args) {
        int sum = 0;
        int item = 0;

        while (item < 5) {
            item ++;
            sum += item;
            if (sum >= 6) break;
        }

        System.out.println("The sum is "+sum);
    }
}
The sum is 6

- Without the if statement, this program calculates the sum of the numbers from 1 to 5. But with the if statement, the loop terminates when the sum becomes greater than or equal to 6.

Page 103, Example 3.6 Demonstrating a continue Statement

// TestContinue.java: Test the continue keyword
public class TestContinue {
    /** Main method */
    public static void main(String[] args) {
        int sum = 0;
        int item = 0;

        while (item < 5) {
            item ++;
            sum += item;
            if (item == 2) continue;
        }

        System.out.println("The sum is "+sum);
    }
}
The sum is 13

- With the if statement, the continue statement is executed when item becomes 2. The continue statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, item is not added to sum when it is 2.
3.6.1 Statement Labels and Breaking with Labels (Optional)

- Every Statement in Java can have an optional label as an identifier. Labels are often used with **break** and **continue** statements.
- You can use a **break** statement with a label to break out of the labeled loop, and a **continue** statement with a label to break out of the current iteration of the labeled loop.
- The break statement given below, for example, breaks out of the outer loop if \((i \times j) > 50\) and transfers control to the statement immediately following the outer loop.

```java
outer:
    for (int i = 1; i < 10; i++) {
        inner:
            for (int j = 1; j < 10; j++) {
                if (i * j > 50)
                    break outer;
                System.out.println(i * j);
            }
    }
```

- If you replace **break outer** with **break** in the preceding statement, the **break** statement would break out of the inner loop and continue to stay inside the outer loop.

- The following **continue** statement breaks out of the inner loop if \((i \times j > 50)\) and starts a new iteration of the outer loop if \(i < 10\) is true after \(i\) us incremented by 1:

```java
outer:
    for (int i = 1; i < 10; i++) {
        inner:
            for (int j = 1; j < 10; j++) {
                if (i * j > 50)
                    continue outer;
                System.out.println(i * j);
            }
    }
```

- If you replace **continue outer** with **continue** in the preceding statement, the **continue** statement would break out of the current iteration of the inner loop if \((i \times j > 50)\) and continue the next iteration of the inner loop if \(j < 10\) is true after \(j\) is incremented by 1.