Chapter 4
Loops

4.1 Introduction

- Loops are structures that control repeated executions of a block of statements.
- Java provides a powerful control structure called a loop, which controls how many times an operation or a sequence of operations is performed in succession.
- Java provides three types of loop statements while loops, do-while loops, and for loops.

4.2 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++;
}
```

**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.
4.2.1 Example: An Advanced Math Learning Tool (Page 97)

- The Math subtraction learning tool program generates just one question for each run. You can use a loop to generate questions repeatedly. This example gives a program that generates ten questions and reports the number of the correct answers after a student answers all ten questions.

```java
import javax.swing.JOptionPane;

public class SubtractionTutorLoop {
    public static void main(String[] args) {
        int correctCount = 0; // Count the number of correct answers
        int count = 0; // Count the number of questions
        long startTime = System.currentTimeMillis();
        String output = "";

        while (count < 10) {
            // 1. Generate two random single-digit integers
            int number1 = (int)(Math.random() * 10);
            int number2 = (int)(Math.random() * 10);

            // 2. If number1 < number2, swap number1 with number2
            if (number1 < number2) {
                int temp = number1;
                number1 = number2;
                number2 = temp;
            }
```
// 3. Prompt the student to answer “what is number1 - number2?”
String answerString = JOptionPane.showInputDialog
    ("What is " + number1 + " - " + number2 + "?");
int answer = Integer.parseInt(answerString);

// 4. Grade the answer and display the result
String replyString;
if (number1 - number2 == answer) {
    replyString = "You are correct!";
    correctCount++;
} else {
    replyString = "Your answer is wrong.
    " + number1 + " - " + number2 + " should be " + (number1 - number2);
    JOptionPane.showMessageDialog(null, replyString);
}

// Increase the count

output += "\n" + number1 + "-" + number2 + "=" + answerString + ((number1 - number2 == answer) ? " correct" : " wrong"));
}

long endTime = System.currentTimeMillis();
long testTime = endTime - startTime;
JOptionPane.showMessageDialog(null,
    "Correct count is " + correctCount + "\nTest time is " +
    testTime / 1000 + " seconds\n" + output);
4.2.2 Controlling a Loop with a Confirmation Dialog

- The preceding example executes the loop ten times. If you want the user to decide whether to take another question, you can use a confirmation dialog to control the loop.
- For example, the following loop continues to execute until the user clicks the No or Cancel button.

```java
int option = 0;
while (option == JOptionPane.YES_OPTION) {
    System.out.println("continue loop");
    option = JOptionPane.showConfirmDialog(null, "Continue?");
}
```

- The value is
  - JOptionPane.YES_OPTION (0) for Yes button,
  - JOptionPane.NO_OPTION (1) for the No button, and
  - JOptionPane.CANCEL_OPTION (2) for Cancel button.
4.2.3 Controlling a Loop with a Sentinel Value

- Often the number of times a loop is executed is not predetermined. You may use an input value to signify the end of the loop. Such a value is known as a sentinel value.
- Write a program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

```java
import javax.swing.JOptionPane;

public class SentinelValue {
    /** Main method */
    public static void main(String[] args) {
        // Read an initial data
        String dataString = JOptionPane.showInputDialog(
            "Enter an int value:
            (the program exits if the input is 0)";
        int data = Integer.parseInt(dataString);

        // Keep reading data until the input is 0
        int sum = 0;
        while (data != 0) {
            sum += data;
            // Read the next data
            dataString = JOptionPane.showInputDialog(
                "Enter an int value:
                (the program exits if the input is 0)";
            data = Integer.parseInt(dataString);
        }

        JOptionPane.showMessageDialog(null, "The sum is " + sum);
    }
}
```

- 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.
4.3 do-while Loop

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

```c
do {
    // Loop body
    Statement(s);
} while (continue-condition);
```

- 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 101), 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 {
            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);
    }
}
```
4.4 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 for loop prints Welcome to Java! 100 times.

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

![Diagram](A) ![Diagram](B)

**FIGURE 3.9** A for loop performs an initial action one, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the loop-continuation-condition evaluates as true.
o The for loop initializes i to 0, then repeatedly executes the println and evaluates i++ if i is less than 100.
o The initial-action, i = 0, initializes the control variable, i.
o The loop-continuation-condition, i < 100, is a Boolean expression.
o The expression is evaluated at the beginning of each iteration.
o 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.
o The action-after-each-iteration, i++, is a statement that adjusts the control variable.
o This statement is executed after each iteration. It increments the control variable.
o Eventually, the value of the control variable forces the loop-continuation-condition to become false.
o 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:

```
for ( ; ; ) {
    // Do something
}
```

Equivalent

```
while (true) {
    // Do something
}
```

(a)  (b)
4.5 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) {
    // Loop body
}
```

```
for (; loop-continuation-condition; ) {
    // 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.

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

```
initial-action;
while (loop-continuation-condition) {
    // Loop body;
    action-after-each-iteration;
}
```

**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:

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

- Similarly, the following loop is also wrong:

```
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**
```
4.6 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.

- **Listing 4.4 Displaying the Multiplication Table (Page 105)**
- Problem: Write a program that uses nested for loops to print a multiplication table.

```java
import javax.swing.JOptionPane;

public class MultiplicationTable {
    /** Main method */
    public static void main(String[] args) {
        // Display the table heading
        String output = "        Multiplication Table
";
        output += "-------------------------------------------------
";
        // Display the number title
        output += "   | ";
        for (int j = 1; j <= 9; j++)
            output += "    " + j;
        output += " 
";
        // 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;
                // Display result
                JOptionPane.showMessageDialog(null, output);
            }
        }
    }
}
```
• 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.
4.7 Minimizing Numerical Errors

- Numeric errors involving floating-point numbers are inevitable.
- LISTING 4.5 (Page 106) 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.
If you change float in the program to double as follows, you should see a slight improvement in precision because a double variable takes 64 bits, whereas a float variable takes 32 bits.

```java
import javax.swing.JOptionPane;

public class TestSum {
    public static void main(String[] args) {
        double sum = 0;

        // Add 0.01, 0.02, ..., 0.99, 1 to sum
        for (double i = 0.01; i <= 1.0; i = i + 0.01)
            sum += i;

        // Display result
        JOptionPane.showMessageDialog(null, "The sum is " + sum);
    }
}
```

Errors commonly occur. There are two ways to fix the problem:

- Minimizing errors by processing large numbers first.
- Using an integer count to ensure that all the numbers are processed.
- To minimize errors, add numbers from 1.0, 0.99, down to 0.1, as follows:

```java
double sum = 0;
double currentValue = 0.01;

for (int count = 0; count < 100; count++) {
    sum += currentValue;
    currentValue += 0.01;
}
```

After this loop, sum is 50.50000000000003.
4.8 Case Studies

- Control statements are fundamental in programming. The ability to write control statements is essential in learning Java programming.
- *If you can write programs using loops, you know how to program!*

4.8.1 Example: Finding the Greatest Common Divisor

- Problem: Write a program that prompts the user to enter two positive integers and finds their **greatest common divisor**.
- LISTING 4.6 (Page 108) Solution: Suppose you enter two integers 4 and 2, their greatest common divisor is 2. Suppose you enter two integers 16 and 24, their greatest common divisor is 8. So, how do you find the greatest common divisor? Let the two input integers be \( n_1 \) and \( n_2 \). You know number 1 is a common divisor, but it may not be the greatest common divisor. So you can check whether \( k \) (for \( k = 2, 3, 4, \) and so on) is a common divisor for \( n_1 \) and \( n_2 \), until \( k \) is greater than \( n_1 \) or \( n_2 \).
String output = "The greatest common divisor for " + n1 + " and " + n2 + " is " + gcd;
JOptionPane.showMessageDialog(null, output);
}
4.9 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.
• LISTING 4.9 (Page 114), Demonstrating a break Statement
  • This program adds the integers from 1 to 20 in this order to sum until sum is greater than or equal to 100.

  Output:
  The number is 14
  The sum is 105

• Without the if statement, the program calculates the sum of the numbers from 1 to 20.

  Output:
  The number is 20
  The sum is 210

  ```java
  public class TestBreak {
      /** Main method */
      public static void main(String[] args) {
          int sum = 0;
          int number = 0;

          while (number < 20) {
              number++;
              sum += number;
              if (sum >= 100) break;
          }

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

  The number is 14
  The sum is 105
• LISTING 4.10 (Page 114), Demonstrating a continue Statement
• This program adds all the integers from 1 to 20 except 10 and 11 to sum.

Output:
The sum is 189

• Without the if statement in the program, all of the numbers are added to sum, even when number is 10 or 11.

Output:
The sum is 210

```java
public class TestContinue {
    /** Main method */
    public static void main(String[] args) {
        int sum = 0;
        int number = 0;

        while (number < 20) {
            number++;
            if (number == 10 || number == 11) continue;
            sum += number;
        }

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

The sum is 189
4.9.1 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 \(i\) is 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.