CHAPTER

5

LOOPS

Objectives

- To write programs for executing statements repeatedly using a `while` loop (§5.2).
- To write loops for the guessing number problem (§5.3).
- To follow the loop design strategy to develop loops (§5.4).
- To control a loop with the user confirmation or a sentinel value (§5.5).
- To obtain large input from a file using input redirection rather than typing from the keyboard (§5.5).
- To write loops using `do-while` statements (§5.6).
- To write loops using `for` statements (§5.7).
- To discover the similarities and differences of three types of loop statements (§5.8).
- To write nested loops (§5.9).
- To learn the techniques for minimizing numerical errors (§5.10).
- To learn loops from a variety of examples (`GCD`, `FutureTuition`, and `Dec2Hex`) (§5.11).
- To implement program control with `break` and `continue` (§5.12).
- To process characters in a string using a loop in a case study for checking palindrome (§5.13).
- To write a program that displays prime numbers (§5.14).

5.1 Introduction

A loop can be used to tell a program to execute statements repeatedly.

problem

Key Point

Suppose you need to display a string (e.g., `Welcome to Java!`) a hundred times. It would be tedious to have to write the following statement a hundred times:

100 times →

```
System.out.println("Programming is fun");
System.out.println("Programming is fun");
...
System.out.println("Programming is fun");
```

So, how do you solve this problem?

loop

Java provides a powerful construct called a *loop* that controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you can simply tell the computer to display a string a hundred times without having to code the print statement a hundred times, as follows:

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

The variable `count` is initially `0`. The loop checks whether `count < 100` is `true`. If so, it executes the loop body to display the message `Welcome to Java!` and increments `count` by `1`. It repeatedly executes the loop body until `count < 100` becomes `false`. When `count < 100` is `false` (i.e., when `count` reaches `100`), the loop terminates, and the next statement after the loop statement is executed.

Loops are constructs that control repeated executions of a block of statements. The concept of looping is fundamental to programming. Java provides three types of loop statements: `while` loops, `do-while` loops, and `for` loops.

5.2 The while Loop

A `while` loop executes statements repeatedly while the condition is true.

Key Point

The syntax for the `while` loop is as follows:

while loop

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

VideoNote

Use while loop

loop body

iteration

loop-continuation-condition

Figure 5.1a shows the `while` loop flowchart. The 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* (or *repetition*) *of the loop*. Each loop contains a `loop-continuation-condition`, a Boolean expression that controls the execution of the body. It is evaluated each time to determine if 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.

The loop for displaying `Welcome to Java!` a hundred times introduced in the preceding section is an example of a `while` loop. Its flowchart is shown in Figure 5.1b.

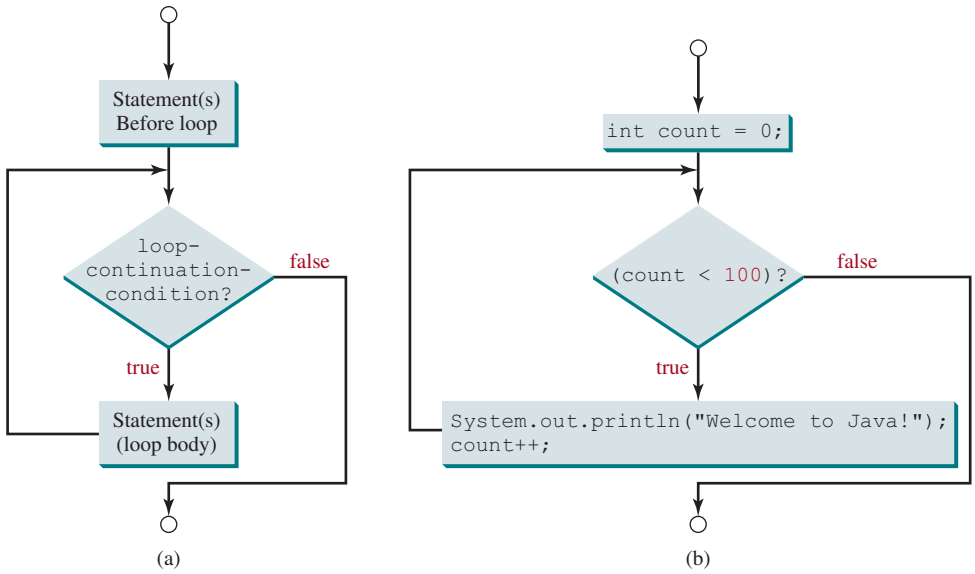


FIGURE 5.1 The `while` loop repeatedly executes the statements in the loop body when the `loop-continuation-condition` evaluates to `true`.

The `loop-continuation-condition` is `count < 100` and the loop body contains two statements in the following code:

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

loop continuation condition

loop body

In this example, you know exactly how many times the loop body needs to be executed because the control variable `count` is used to count the number of iterations. This type of loop is known as a *counter-controlled loop.*

counter-controlled loop

Note

The `loop-continuation-condition` must always appear inside the parentheses. The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

Here is another example to help understand how a loop works.

```
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
  i++;
}
System.out.println("sum is " + sum); // sum is 45
```

If `i < 10` is `true`, the program adds `i` to `sum`. Variable `i` is initially set to `1`, then is incremented to `2`, `3`, and up to `10`. When `i` is `10`, `i < 10` is `false`, so the loop exits. Therefore, the sum is `1 + 2 + 3 + ... + 9 = 45`.

What happens if the loop is mistakenly written as follows?

```
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
}
```

This loop is infinite, because `i` is always `1` and `i < 10` will always be `true`.

Note

infinite loop

Make sure that the `loop-continuation-condition` eventually becomes `false` so that the loop will terminate. A common programming error involves *infinite loops* (i.e., the loop runs forever). If your program takes an unusually long time to run and does not stop, it may have an infinite loop. If you are running the program from the command window, press *CTRL+C* to stop it.

Caution

off-by-one error

Programmers often make the mistake of executing a loop one more or less time. This is commonly known as the *off-by-one error*. For example, the following loop displays `Welcome to Java` 101 times rather than 100 times. The error lies in the condition, which should be `count < 100` rather than `count <= 100`.

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

Recall that Listing 3.1, AdditionQuiz.java, gives a program that prompts the user to enter an answer for a question on addition of two single digits. Using a loop, you can now rewrite the program to let the user repeatedly enter a new answer until it is correct, as given in Listing 5.1.

LISTING 5.1 RepeatAdditionQuiz.java

```
1  import java.util.Scanner;
2
3  public class RepeatAdditionQuiz {
4    public static void main(String[] args) {
5      int number1 = (int)(Math.random() * 10);
6      int number2 = (int)(Math.random() * 10);
7
8      // Create a Scanner
9      Scanner input = new Scanner(System.in);
10
11     System.out.print(
12       "What is " + number1 + " + " + number2 + "? ");
13     int answer = input.nextInt();
14
15     while (number1 + number2 != answer)  {
16       System.out.print("Wrong answer. Try again. What is "
17         + number1 + " + " + number2 + "? ");
18       answer = input.nextInt();
19     }
20
21     System.out.println("You got it!");
22   }
23 }
```

generate number1
generate number2
show question
get first answer
check answer
read an answer

```
What is 5 + 9? 12 Enter
Wrong answer. Try again. What is 5 + 9? 34 Enter
Wrong answer. Try again. What is 5 + 9? 14 Enter
You got it!
```

The loop in lines 15–19 repeatedly prompts the user to enter an `answer` when `number1 + number2 != answer` is `true`. Once `number1 + number2 != answer` is `false`, the loop exits.

5.2.1 Analyze the following code. Is `count < 100` always `true`, always `false`, or sometimes `true` or sometimes `false` at Point A, Point B, and Point C?


```
int count = 0;
while (count < 100) {
  // Point A
  System.out.println("Welcome to Java!");
  count++;
  // Point B
}
// Point C
```

5.2.2 How many times are the following loop bodies repeated? What is the output of each loop?

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i);
```

(a)

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i++);
```

(b)

```
int i = 1;
while (i < 10)
  if ((i++) % 2 == 0)
    System.out.println(i);
```

(c)

5.2.3 What is the output of the following code? Explain the reason.

```
int x = 80000000;

while (x > 0)
  x++;

System.out.println("x is " + x);
```

5.3 Case Study: Guessing Numbers

This case study generates a random number and lets the user repeatedly guess a number until it is correct.

VideoNote

Guess a number

The problem is to guess what number a computer has in mind. You will write a program that randomly generates an integer between `0` and `100`, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can make the next guess intelligently. Here is a sample run:

```
Guess a magic number between 0 and 100
Enter your guess: 50 Enter
Your guess is too high
Enter your guess: 25 Enter
Your guess is too low
Enter your guess: 42 Enter
Your guess is too high
Enter your guess: 39 Enter
Yes, the number is 39
```

intelligent guess

The magic number is between **0** and **100**. To minimize the number of guesses, enter **50** first. If your guess is too high, the magic number is between **0** and **49**. If your guess is too low, the magic number is between **51** and **100**. Thus, you can eliminate half of the numbers from further consideration after one guess.

think before coding

How do you write this program? Do you immediately begin coding? No. It is important to *think before coding*. Think how you would solve the problem without writing a program. You need first to generate a random number between **0** and **100**, inclusive, then to prompt the user to enter a guess, then to compare the guess with the random number.

code incrementally

It is a good practice to *code incrementally* one step at a time. For programs involving loops, if you don't know how to write a loop right away, you may first write the code for executing the loop one time, then figure out how to repeatedly execute the code in a loop. For this program, you may create an initial draft, as given in Listing 5.2.

LISTING 5.2 GuessNumberOneTime.java

```
1  import java.util.Scanner;
2
3  public class GuessNumberOneTime {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a magic number between 0 and 100");
10
11     // Prompt the user to guess the number
12     System.out.print("\nEnter your guess: ");
13     int guess = input.nextInt();
14
15     if (guess == number)
16       System.out.println("Yes, the number is " + number);
17     else if (guess > number)
18       System.out.println("Your guess is too high");
19     else
20       System.out.println("Your guess is too low");
21   }
22 }
```

generate a number

enter a guess

correct guess

too high

too low

When you run this program, it prompts the user to enter a guess only once. To let the user enter a guess repeatedly, you may wrap the code in lines 11–20 in a loop as follows:

```
while (true) {
  // Prompt the user to guess the number
  System.out.print("\nEnter your guess: ");
  guess = input.nextInt();

  if (guess == number)
    System.out.println("Yes, the number is " + number);
  else if (guess > number)
    System.out.println("Your guess is too high");
  else
    System.out.println("Your guess is too low");
} // End of loop
```

This loop repeatedly prompts the user to enter a guess. However, this loop is not correct, because it never terminates. When **guess** matches **number**, the loop should end. Thus, the loop can be revised as follows:

```
while (guess != number) {
  // Prompt the user to guess the number
```

```
    System.out.print("\nEnter your guess: ");
    guess = input.nextInt();

    if (guess == number)
      System.out.println("Yes, the number is " + number);
    else if (guess > number)
      System.out.println("Your guess is too high");
    else
      System.out.println("Your guess is too low");
  } // End of loop
```

The complete code is given in Listing 5.3.

LISTING 5.3 GuessNumber.java

```
1  import java.util.Scanner;
2
3  public class GuessNumber {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);                    generate a number
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a magic number between 0 and 100");
10
11     int guess = -1;
12     while (guess != number) {
13       // Prompt the user to guess the number
14       System.out.print("\nEnter your guess: ");
15       guess = input.nextInt();                                  enter a guess
16
17       if (guess == number)
18         System.out.println("Yes, the number is " + number);
19       else if (guess > number)
20         System.out.println("Your guess is too high");          too high
21       else
22         System.out.println("Your guess is too low");           too low
23     } // End of loop
24   }
25 }
```

	line#	number	guess	output
	6	39		
	11		-1	
iteration 1	15		50	
	20			Your guess is too high
iteration 2	15		25	
	22			Your guess is too low
iteration 3	15		42	
	20			Your guess is too high
iteration 4	15		39	
	18			Yes, the number is 39

The program generates the magic number in line 6 and prompts the user to enter a guess continuously in a loop (lines 12–23). For each guess, the program checks whether the guess is

correct, too high, or too low (lines 17–22). When the guess is correct, the program exits the loop (line 12). Note that `guess` is initialized to `-1`. Initializing it to a value between `0` and `100` would be wrong, because that could be the number to be guessed.

5.3.1 What is wrong if `guess` is initialized to `0` in line 11 in Listing 5.3?

5.4 Loop Design Strategies

The key to designing a loop is to identify the code that needs to be repeated and write a condition for terminating the loop.

Writing a correct loop is not an easy task for novice programmers. Consider three steps when writing a loop.

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop as follows:

```
while (true) {
  Statements;
}
```

Step 3: Code the `loop-continuation-condition` and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition) {
  Statements;
  Additional statements for controlling the loop;
}
```

VideoNote

Multiple subtraction quiz

The Math subtraction learning tool program in Listing 3.3, SubtractionQuiz.java, generates just one question for each run. You can use a loop to generate questions repeatedly. How do you write the code to generate five questions? Follow the loop design strategy. First, identify the statements that need to be repeated. These are the statements for obtaining two random numbers, prompting the user with a subtraction question, and grading the question. Second, wrap the statements in a loop. Third, add a loop control variable and the `loop-continuation-condition` to execute the loop five times.

Listing 5.4 gives a program that generates five questions and, after a student answers all five, reports the number of correct answers. The program also displays the time spent on the test and lists all the questions.

LISTING 5.4 SubtractionQuizLoop.java

```
1  import java.util.Scanner;
2
3  public class SubtractionQuizLoop {
4    public static void main(String[] args) {
5      final int NUMBER_OF_QUESTIONS = 5; // Number of questions
6      int correctCount = 0; // Count the number of correct answers
7      int count = 0; // Count the number of questions
8      long startTime = System.currentTimeMillis();
9      String output = " "; // output string is initially empty
10     Scanner input = new Scanner(System.in);
11
12     while (count < NUMBER_OF_QUESTIONS) {
13       // 1. Generate two random single-digit integers
14       int number1 = (int)(Math.random() * 10);
15       int number2 = (int)(Math.random() * 10);
16
17       // 2. If number1 < number2, swap number1 with number2
18       if (number1 < number2) {
```

get start time

loop

```
19          int temp = number1;
20          number1 = number2;
21          number2 = temp;
22        }
23
24        // 3. Prompt the student to answer "What is number1 - number2?"
25        System.out.print(                                          display a question
26          "What is " + number1 + " - " + number2 + "? ");
27        int answer = input.nextInt();
28
29        // 4. Grade the answer and display the result
30        if (number1 - number2 == answer) {                         grade an answer
31          System.out.println("You are correct!");
32          correctCount++; // Increase the correct answer count      increase correct count
33        }
34        else
35          System.out.println("Your answer is wrong.\n" + number1
36            + " - " + number2 + " should be " + (number1 - number2));
37
38        // Increase the question count
39        count++;                                                   increase control variable
40
41        output += "\n" + number1 + "-" + number2 + "=" + answer +  prepare output
42          ((number1 - number2 == answer) ? " correct": " wrong");
43      }                                                            end loop
44
45      long endTime = System.currentTimeMillis();                   get end time
46      long testTime = endTime - startTime;                         test time
47
48      System.out.println("Correct count is " + correctCount +     display result
49        "\nTest time is " + testTime / 1000 + " seconds\n" + output);
50    }
51  }
```

```
What is 9 - 2?  7 Enter
You are correct!

What is 3 - 0?  3 Enter
You are correct!

What is 3 - 2?  1 Enter
You are correct!

What is 7 - 4?  4 Enter
Your answer is wrong.
7 - 4 should be 3

What is 7 - 5?  4 Enter
Your answer is wrong.
7 - 5 should be 2

Correct count is 3
Test time is 1021 seconds

9-2=7 correct
3-0=3 correct
3-2=1 correct
7-4=4 wrong
7-5=4 wrong
```

The program uses the control variable `count` to control the execution of the loop. `count` is initially `0` (line 7) and is increased by `1` in each iteration (line 39). A subtraction question is displayed and processed in each iteration. The program obtains the time before the test starts in line 8 and the time after the test ends in line 45, then computes the test time in line 46. The test time is in milliseconds and is converted to seconds in line 49.

5.4.1 Revise the code using the `System.nanoTime()` to measure the time in nano seconds.

5.5 Controlling a Loop with User Confirmation or a Sentinel Value

It is a common practice to use a sentinel value to terminate the input.

The preceding example executes the loop five times. If you want the user to decide whether to continue, you can offer a user *confirmation*. The template of the program can be coded as follows:

```
char continueLoop = 'Y';
while (continueLoop == 'Y') {
  // Execute the loop body once
  ...
  // Prompt the user for confirmation
  System.out.print("Enter Y to continue and N to quit: ");
  continueLoop = input.getLine().charAt(0);
}
```

You can rewrite the program given in Listing 5.4 with user confirmation to let the user decide whether to advance to the next question.

sentinel value

sentinel-controlled loop

Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a *sentinel value*, signifies the end of the input. A loop that uses a sentinel value to control its execution is called a *sentinel-controlled loop*.

Listing 5.5 gives a program that reads and calculates the sum of an unspecified number of integers. The input `0` signifies the end of the input. Do you need to declare a new variable for each input value? No. Just use one variable named `data` (line 12) to store the input value, and use a variable named `sum` (line 15) to store the total. Whenever a value is read, assign it to `data` and, if it is not zero, add it to `sum` (line 17).

LISTING 5.5 SentinelValue.java

```
1  import java.util.Scanner;
2
3  public class SentinelValue {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Read an initial data
10     System.out.print(
11       "Enter an integer (the input ends if it is 0): ");
12     int data = input.nextInt();
13
14     // Keep reading data until the input is 0
15     int sum = 0;
16     while (data != 0) {
17       sum += data;
18
19       // Read the next data
20       System.out.print(
```

input

loop

```
21          "Enter an integer (the input ends if it is 0): ");
22        data = input.nextInt();
23      }                                                          end of loop
24
25      System.out.println("The sum is " + sum);                   display result
26    }
27  }
```

```
Enter an integer (the input ends if it is 0): 2 Enter
Enter an integer (the input ends if it is 0): 3 Enter
Enter an integer (the input ends if it is 0): 4 Enter
Enter an integer (the input ends if it is 0): 0 Enter
The sum is 9
```

	line#	data	sum	output
	12	2		
	15		0	
iteration 1	17		2	
	22	3		
iteration 2	17		5	
	22	4		
iteration 3	17		9	
	22	0		
	25			The sum is 9

If `data` is not `0`, it is added to `sum` (line 17) and the next item of input data is read (lines 20–22). If `data` is `0`, the loop body is no longer executed and the `while` loop terminates. The input value `0` is the sentinel value for this loop. Note if the first input read is `0`, the loop body never executes, and the resulting sum is `0`.

Caution

Don't use floating-point values for equality checking in a loop control. Because floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results.

Consider the following code for computing `1 + 0.9 + 0.8 + ... + 0.1`:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

Variable `item` starts with `1` and is reduced by `0.1` every time the loop body is executed. The loop should terminate when `item` becomes `0`. However, there is no guarantee that item will be exactly `0`, because the floating-point arithmetic is approximated. This loop seems okay on the surface, but it is actually an infinite loop. numeric error

In the preceding example, if you have a large number of data to enter, it would be cumbersome to type from the keyboard. You can store the data separated by whitespaces in a text file, say **input.txt**, and run the program using the following command:

```
java SentinelValue < input.txt
```

input redirection

This command is called *input redirection*. The program takes the input from the file **input.txt** rather than having the user type the data from the keyboard at runtime. Suppose the contents of the file are as follows:

```
2 3 4 5 6 7 8 9 12 23 32
23 45 67 89 92 12 34 35 3 1 2 4 0
```

The program should get `sum` to be `518`.

output redirection

Similarly, there is *output redirection*, which sends the output to a file rather than displaying it on the console. The command for output redirection is

```
java ClassName > output.txt
```

Input and output redirections can be used in the same command. For example, the following command gets input from **input.txt** and sends output to **output.txt**:

```
java SentinelValue < input.txt > output.txt
```

Try running the program to see what contents are in **output.txt**.

When reading data through input redirection, you can invoke `input.hasNext()` to detect the end of input. For example, the following code reads all `int` value from the input and displays their total.

```
import java.util.Scanner;

public class TestEndOfInput {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    int sum = 0;

    while (input.hasNext ()) {
      sum += input.nextInt();
    }

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

If there is no more input in the file, `input.hasNext()` will return `false`.

Note

If you enter the input from the command window, you can end the input by pressing ENTER and then CTRL+Z, and then pressing ENTER again. In this case, `input.hasNext()` will return `false`.

5.5.1 Suppose the input is `2 3 4 5 0`. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int number, max;
    number = input.nextInt(); max = number;

    while (number != 0) {
      number = input.nextInt();
      if (number > max)
        max = number;
    }

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

5.6 The do-while Loop

A **do-while** *loop is the same as a* **while** *loop except that it executes the loop body first then checks the loop continuation condition.*

VideoNote

Use do-while loop

do-while loop

The **do-while** *loop* is a variation of the **while** loop. Its syntax is as follows:

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

Its execution flowchart is shown in Figure 5.2a.

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. For example, the following **while** loop statement

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

can be written using a **do-while** loop as follows:

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

The flowchart of this **do-while** loop is shown in Figure 5.2b.

The 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 is executed. In the case of a **do-while** loop, the loop body is executed at least once. You can write a loop using either the **while** loop or the **do-while** loop. Sometimes one is a more convenient choice than the other. For example, you can rewrite the **while** loop in Listing 5.5 using a **do-while** loop, as given in Listing 5.6.

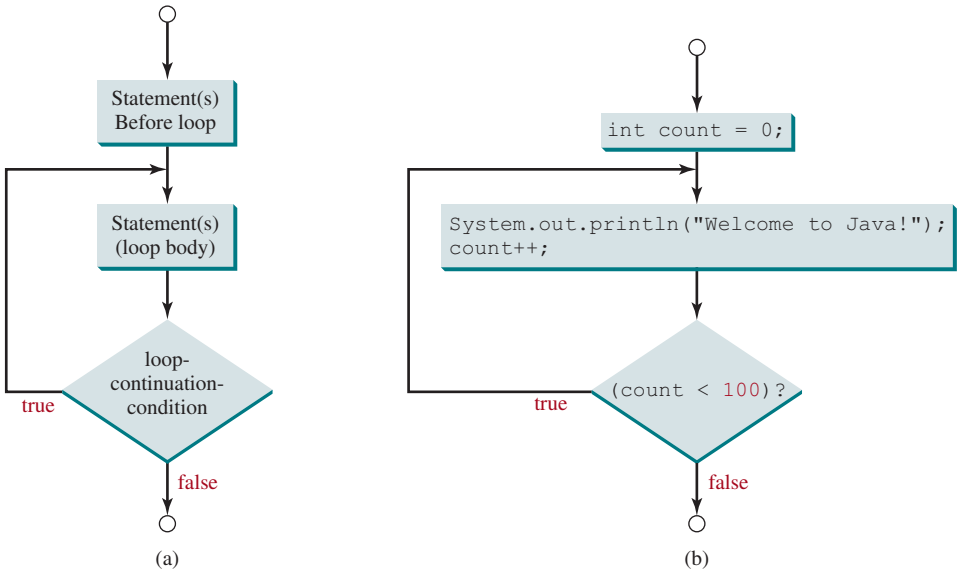


FIGURE 5.2 The **do-while** loop executes the loop body first then checks the **loop-continuation-condition** to determine whether to continue or terminate the loop.

LISTING 5.6 TestDoWhile.java

```
1  import java.util.Scanner;
2
3  public class TestDoWhile {
4    /** Main method */
5    public static void main(String[] args) {
6      int data;
7      int sum = 0;
8
9      // Create a Scanner
10     Scanner input = new Scanner(System.in);
11
12     // Keep reading data until the input is 0
13     do {                                          loop
14       // Read the next data
15       System.out.print(
16         "Enter an integer (the input ends if it is 0): ");
17       data = input.nextInt();
18
19       sum += data;
20     } while (data != 0);                          end loop
21
22     System.out.println("The sum is " + sum);
23   }
24 }
```

```
Enter an integer (the input ends if it is 0): 3 Enter
Enter an integer (the input ends if it is 0): 5 Enter
Enter an integer (the input ends if it is 0): 6 Enter
Enter an integer (the input ends if it is 0): 0 Enter
The sum is 14
```

Tip

Use a **do-while** loop if you have statements inside the loop that must be executed *at least once*, as in the case of the **do-while** loop in the preceding **TestDoWhile** program. These statements must appear before the loop as well as inside it if you use a **while** loop.

5.6.1 Suppose the input is **2 3 4 5 0**. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, max;
    number = input.nextInt();
    max = number;

    do {
      number = input.nextInt();
      if (number > max)
        max = number;
    } while (number != 0);
```

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

5.6.2 What are the differences between a `while` loop and a `do-while` loop? Convert the following `while` loop into a `do-while` loop:

```
Scanner input = new Scanner(System.in);
int sum = 0;
System.out.println("Enter an integer " +
   "(the input ends if it is 0)");
int number = input.nextInt();
while (number != 0) {
  sum += number;
  System.out.println("Enter an integer " +
     "(the input ends if it is 0)");
  number = input.nextInt();
}
```

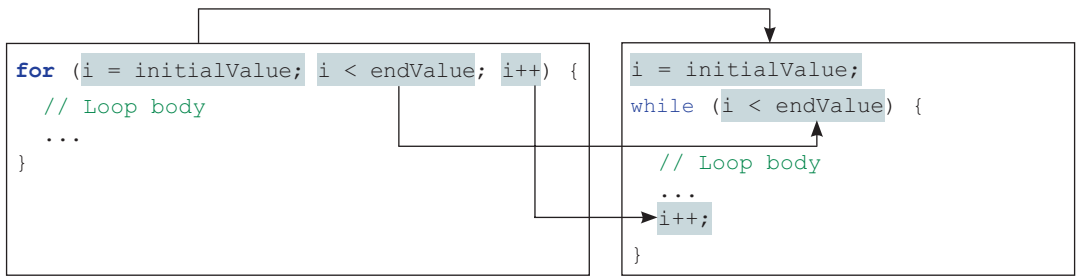
5.7 The for Loop

A `for` loop has a concise syntax for writing loops.

Often you write a loop in the following common form:

```
i = initialValue; // Initialize loop control variable
while (i < endValue) {
  // Loop body
  ...
  i++; // Adjust loop control variable
}
```

This loop is intuitive and easy for beginners to grasp. However, programmers often forget to adjust the control variable, which leads to an infinite loop. A `for` loop can be used to avoid the potential error and simplify the preceding loop as shown in (a) below. In general, the syntax for a for loop is as shown in (a), which is equivalent to (b).



In general, the syntax of a `for` *loop* is as follows:

for loop

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

The flowchart of the `for` loop is shown in Figure 5.3a.

The `for` loop statement starts with the keyword `for`, followed by a pair of parentheses enclosing the control structure of the loop. This structure consists of `initial-action`, `loop-continuation-condition`, and `action-after-each-iteration`. The control structure is

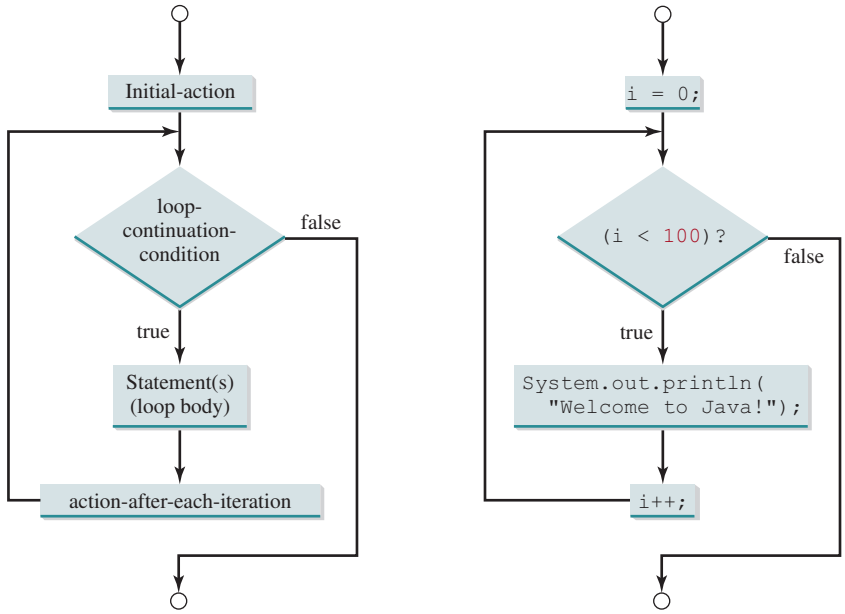


FIGURE 5.3 A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the `loop-continuation-condition` evaluates to `true`.

followed by the loop body enclosed inside braces. The `initial-action`, `loop-continuation-condition`, and `action-after-each-iteration` are separated by semicolons.

control variable

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. For example, the following `for` loop prints `Welcome to Java!` a hundred times:

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

The flowchart of the statement is shown in Figure 5.3b. The `for` loop initializes `i` to `0`, then repeatedly executes the `println` statement and evaluates `i++` while `i` is less than `100`.

initial-action

The `initial-action`, `i = 0`, initializes the control variable, `i`. The `loop-continuation-condition`, `i < 100`, is a Boolean expression. The expression is evaluated right after the initialization and at the beginning of each iteration. If this condition is `true`, the loop body is executed. If it is `false`, the loop terminates and the program control turns to the line following the loop.

action-after-each-iteration

The `action-after-each-iteration`, `i++`, is a statement that adjusts the control variable. This statement is executed after each iteration and increments the control variable. Eventually, the value of the control variable should force the `loop-continuation-condition` to become `false`; otherwise, the loop is infinite.

The loop control variable can be declared and initialized in the `for` loop. Here is an example:

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

omitting braces

If there is only one statement in the loop body, as in this example, the braces can be omitted.

declare control variable

Tip

The control variable must be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is a good programming practice to declare it in the **initial-action** of the **for** loop. If the variable is declared inside the loop control structure, it cannot be referenced outside the loop. In the preceding code, for example, you cannot reference **i** outside the **for** loop, because it is declared inside the **for** loop.

for loop variations

Note

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

```
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. For example:

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

This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action, and increment or decrement the control variable as an action after each iteration.

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 the same as in (b). To avoid confusion, though, it is better to use the equivalent loop in (c).

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

(a)

Equivalent

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

(b)

Equivalent

This is better

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

(c)

5.7.1 Do the following two loops result in the same value in **sum**?

Check Point

```
for (int i = 0; i < 10; ++i) {
  sum += i;
}
```

(a)

```
for (int i = 0; i < 10; i++) {
  sum += i;
}
```

(b)

5.7.2 What are the three parts of a **for** loop control? Write a **for** loop that prints the numbers from **1** to **100**.

5.7.3 Suppose the input is **2 3 4 5 0**. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, sum = 0, count;
    for (count = 0; count < 5; count++) {
      number = input.nextInt();
      sum += number;
    }
```

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

5.7.4 What does the following statement do?

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

5.7.5 If a variable is declared in a `for` loop control, can it be used after the loop exits?

5.7.6 Convert the following `for` loop statement to a `while` loop and to a `do-while` loop:

```
long sum = 0;
for (int i = 0; i <= 1000; i++)
  sum = sum + i;
```

5.7.7 Count the number of iterations in the following loops.

```
int count = 0;
while (count < n) {
  count++;
}
```

(a)

```
for (int count = 0;
  count <= n; count++) {
}
```

(b)

```
int count = 5;
while (count < n) {
  count++;
}
```

(c)

```
int count = 5;
while (count < n) {
  count = count + 3;
}
```

(d)

5.8 Which Loop to Use?

You can use a `for` loop, a `while` loop, or a `do-while` loop, whichever is convenient.

The `while` loop and `do-while` loop are easier to learn than the `for` loop. However, you will learn the `for` loop quickly after some practice. A `for` loop places control variable initialization, loop continuation condition, and adjustment after each iteration all together. It is more concise and enables you to write the code with less errors than the other two loops.

pretest loop
posttest loop

The `while` loop and `for` loop are called *pretest loops* because the continuation condition is checked before the loop body is executed. The `do-while` loop is called a *posttest loop* because the condition is checked after the loop body is executed. The three forms of loop statements—`while`, `do-while`, 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 `for` loop in (b).

```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; ) {
  // Loop body
}
```

(b)

A `for` loop in (a) in the next figure can generally be converted into the `while` loop in (b) except in certain special cases (see CheckPoint Question 5.12.2 in Section 5.12 for such a case).

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

(a)

Equivalent

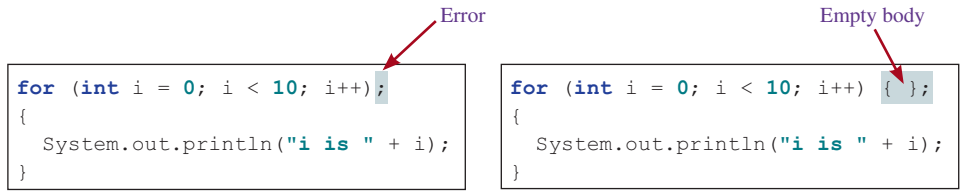
```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

Use the loop statement that is most intuitive and comfortable for you. In general, a `for` loop may be used if the number of repetitions is known in advance, as, for example, when you need to display a message a hundred times. A `while` loop may be used if the number of repetitions is not fixed, 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 the continuation condition is tested.

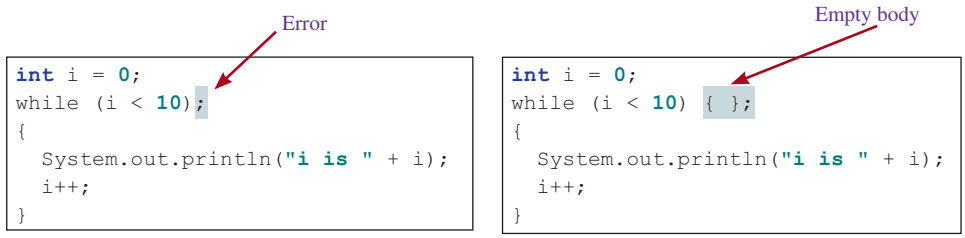
Caution

Adding a semicolon at the end of the `for` clause before the loop body is a common mistake, as shown below in (a). In (a), the semicolon signifies the end of the loop prematurely. The loop body is actually empty, as shown in (b). (a) and (b) are equivalent. Both are incorrect.



(a) (b)

Similarly, the loop in (c) is also wrong. (c) is equivalent to (d). Both are incorrect.



(c) (d)

These errors often occur when you use the next-line block style. Using the end-of-line block style can avoid errors of this type.

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

```
int i = 0;
do {
  System.out.println("i is " + i);
  i++;
} while (i < 10);
```

This is correct.

5.8.1 Can you convert a `for` loop to a `while` loop? List the advantages of using `for` loops.

5.8.2 Can you always convert a `while` loop into a `for` loop? Convert the following `while` loop into a `for` loop:

```
int i = 1;
int sum = 0;
while (sum <  10000) {
  sum = sum + i;
  i++;
}
```

5.8.3 Identify and fix the errors in the following code:

```
1  public class Test {
2    public void main(String[] args) {
3      for (int i = 0; i < 10; i++);
4        sum += i;
5
6      if (i < j);
7        System.out.println(i)
8      else
9        System.out.println(j);
10
11     while (j < 10);
12     {
13       j++;
14     }
15
16     do {
17       j++;
18     } while (j < 10)
19   }
20 }
```

5.8.4 What is wrong with the following programs?

```
1  public class ShowErrors {
2    public static void main(String[] args) {
3      int i = 0;
4      do  {
5        System.out.println(i + 4);
6        i++;
7      }
8      while (i < 10)
8    }
9  }
```

(a)

```
1  public class ShowErrors {
2    public static void main(String[] args) {
3      for (int i = 0; i < 10; i++);
4        System.out.println(i + 4);
5    }
6  }
```

(b)

5.9 Nested Loops

A loop can be nested inside another loop.

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 started anew.

nested loop

Listing 5.7 presents a program that uses nested `for` loops to display a multiplication table.

LISTING 5.7 MultiplicationTable.java

```
1  public class MultiplicationTable {
2    /** Main method */
3    public static void main(String[] args) {
4      // Display the table heading
5      System.out.println("         Multiplication Table");
6
7      // Display the number title
8      System.out.print("   ");
9      for (int j = 1; j <= 9; j++)
10       System.out.print("   " + j);
```

table title

```
11
12      System.out.println("\n - - - - - - - - - - -— - - - - - - - - - -");
13
14      // Display table body
15      for (int i = 1; i <= 9; i++) {                          outer loop
16        System.out.print(i + " | ");
17        for (int j = 1; j <= 9; j++) {                        inner loop
18          // Display the product and align properly
19          System.out.printf("%4d", i * j);
20        }
21        System.out.println();
22      }
23    }
24  }
```

```
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
```

The program displays a title (line 5) on the first line in the output. The first `for` loop (lines 9 and 10) displays the numbers `1–9` on the second line. A dashed (`–`) line is displayed on the third line (line 12).

The next loop (lines 15–22) is a nested `for` 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`.

Note

Be aware that a nested loop may take a long time to run. Consider the following loop nested in three levels:

```
for (int i = 0; i < 10000; i++)
  for (int j = 0; j < 10000; j++)
    for (int k = 0; k < 10000; k++)
      Perform an action
```

The action is performed one trillion times. If it takes 1 microsecond to perform the action, the total time to run the loop would be more than 277 hours. Note 1 microsecond is one-millionth (10^{-6}) of a second.

5.9.1 How many times is the `println` statement executed?


```
for (int i = 0; i < 10; i++)
  for (int j = 0; j < i; j++)
    System.out.println(i * j)
```

5.9.2 Show the output of the following programs. (*Hint*: Draw a table and list the variables in the columns to trace these programs.)

```
public class Test {
  public static void main(String[] args) {
    for (int i = 1; i < 5; i++) {
      int j = 0;
      while (j < i) {
        System.out.print(j + " ");
        j++;
      }
    }
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int i = 0;
    while (i < 5) {
      for (int j = i; j > 1; j--)
        System.out.print(j + " ");
      System.out.println("****");
      i++;
    }
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    int i = 5;
    while (i >= 1) {
      int num = 1;
      for (int j = 1; j <= i; j++) {
        System.out.print(num + "xxx");
        num *= 2;
      }

      System.out.println();
      i--;
    }
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    int i = 1;
    do {
      int num = 1;
      for (int j = 1; j <= i; j++) {
        System.out.print(num + "G");
        num += 2;
      }

      System.out.println();
      i++;
    } while (i <= 5);
  }
}
```

(d)

5.10 Minimizing Numeric Errors

Using floating-point numbers in the loop continuation condition may cause numeric errors.

Key Point

VideoNote

Minimize numeric errors

Numeric errors involving floating-point numbers are inevitable, because floating-point numbers are represented in approximation in computers by nature. This section discusses how to minimize such errors through an example.

Listing 5.8 presents an example summing 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.

LISTING 5.8 TestSum.java

loop

```
1  public class TestSum {
2    public static void main(String[] args) {
3      // Initialize sum
4      float sum = 0;
5
6      // Add 0.01, 0.02, ..., 0.99, 1 to sum
7      for (float i = 0.01f; i <= 1.0f; i = i + 0.01f)
8        sum += i;
9
10     // Display result
```

```
11      System.out.println("The sum is " + sum);
12    }
13  }
```

```
The sum is 50.499985
```

The `for` loop (lines 7 and 8) repeatedly adds the control variable `i` to `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 `for` loop initial action can be any statement, but it is often used to initialize a control variable. From this example, you can see a control variable can be a `float` type. In fact, it can be any data type.

double precision

The exact `sum` should be `50.50`, but the answer is `50.499985`. The result is imprecise because computers use a fixed number of bits to represent floating-point numbers, and thus they cannot represent some floating-point numbers exactly. If you change `float` in the program to `double`, as follows, you should see a slight improvement in precision, because a `double` variable holds 64 bits, whereas a `float` variable holds 32 bits.

```
// Initialize sum
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;
```

numeric error

However, you will be stunned to see the result is actually `49.50000000000003`. What went wrong? If you display `i` for each iteration in the loop, you will see that the last `i` is slightly larger than `1` (not exactly `1`). This causes the last `i` not to be added into `sum`. The fundamental problem is the floating-point numbers are represented by approximation. To fix the problem, use an integer count to ensure all the numbers are added to `sum`. Here is the new loop:

```
double currentValue = 0.01;

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

After this loop, `sum` is `50.50000000000003`. This loop adds the numbers from smallest to biggest. What happens if you add numbers from biggest to smallest (i.e., `1.0`, `0.99`, `0.98`,. . . , `0.02`, `0.01` in this order) is as follows:

```
double currentValue = 1.0;

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

avoiding numeric error

After this loop, `sum` is `50.49999999999995`. Adding from biggest to smallest is less accurate than adding from smallest to biggest. This phenomenon is an artifact of the finite-precision arithmetic. Adding a very small number to a very big number can have no effect if the result requires more precision than the variable can store. For example, the inaccurate result of `100000000.0 + 0.000000001` is `100000000.0`. To obtain more accurate results, carefully select the order of computation. Adding smaller numbers before bigger numbers to sum is one way to minimize errors.

5.11 Case Studies

Loops are fundamental in programming. The ability to write loops is essential in learning Java programming.

If you can write programs using loops, you know how to program! For this reason, this section presents three additional examples of solving problems using loops.

5.11.1 Case Study: Finding the Greatest Common Divisor

gcd

The greatest common divisor (gcd) of the two integers **4** and **2** is **2**. The greatest common divisor of the two integers **16** and **24** is **8**. How would you write this program to find the greatest common divisor? Would you immediately begin to write the code? No. It is important to

think before you code

think before you code. Thinking enables you to generate a logical solution for the problem without concern about how to write the code.

Let the two input integers be **n1** and **n2**. You know that number **1** is a common divisor, but it may not be the greatest common divisor. Therefore, you can check whether **k** (for **k** = **2**, **3**, **4**, and so on) is a common divisor for **n1** and **n2**, until **k** is greater than **n1** or **n2**. Store the common divisor in a variable named **gcd**. Initially, **gcd** is **1**. Whenever a new common divisor is found, it becomes the new **gcd**. When you have checked all the possible common divisors from **2** up to **n1** or **n2**, the value in variable **gcd** is the greatest common divisor.

logical solution

Once you have a *logical solution*, type the code to translate the solution into a Java program as follows:

```
int gcd = 1; // Initial gcd is 1
int k = 2; // Possible gcd

while (k <= n1 && k <= n2) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k; // Update gcd
  k++; // Next possible gcd
}

// After the loop, gcd is the greatest common divisor for n1 and n2
```

Listing 5.9 presents the program that prompts the user to enter two positive integers and finds their greatest common divisor.

LISTING 5.9 GreatestCommonDivisor.java

```
1  import java.util.Scanner;
2
3  public class GreatestCommonDivisor {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter two integers
10     System.out.print("Enter first integer: ");
11     int n1 = input.nextInt();
12     System.out.print("Enter second integer: ");
13     int n2 = input.nextInt();
14
15     int gcd = 1; // Initial gcd is 1
16     int k = 2; // Possible gcd
17     while (k <= n1 && k <= n2) {
18       if (n1 % k == 0 && n2 % k == 0)
19         gcd = k; // Update gcd
20       k++;
```

input

input

gcd

check divisor

```
21      }
22
23      System.out.println("The greatest common divisor for " + n1 +
24        " and " + n2 + " is " + gcd);
25    }
26  }
```

output

```
Enter first integer: 125 Enter
Enter second integer: 2525 Enter
The greatest common divisor for 125 and 2525 is 25
```

Translating a logical solution to Java code is not unique. For example, you could use a `for` loop to rewrite the code as follows:

think before you type

```
for (int k = 2; k <= n1 && k <= n2; k++) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k;
}
```

A problem often has multiple solutions, and the gcd problem can be solved in many ways. Programming Exercise 5.14 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm (see Section 22.6).

multiple solutions

You might think that a divisor for a number `n1` cannot be greater than `n1 / 2` and would attempt to improve the program using the following loop:

erroneous solutions

```
for (int k = 2; k <= n1 / 2 && k <= n2 / 2; k++) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k;
}
```

This revision is wrong. Can you find the reason? See Checkpoint Question 5.11.1 for the answer.

5.11.2 Case Study: Predicting the Future Tuition

Suppose the tuition for a university is `$10,000` this year and tuition increases `7%` every year. In how many years will the tuition be doubled?

Before you can write a program to solve this problem, first consider how to solve it by hand. The tuition for the second year is the tuition for the first year * `1.07`. The tuition for a future year is the tuition of its preceding year * `1.07`. Thus, the tuition for each year can be computed as follows:

think before you code

```
double tuition = 10000; int year = 0; // Year 0
tuition = tuition * 1.07; year++;     // Year 1
tuition = tuition * 1.07; year++;     // Year 2
tuition = tuition * 1.07; year++;     // Year 3
...
```

Keep computing the tuition for a new year until it is at least `20000`. By then, you will know how many years it will take for the tuition to be doubled. You can now translate the logic into the following loop:

```
double tuition = 10000; // Year 0
int year = 0;
while (tuition < 20000) {
  tuition = tuition * 1.07;
  year++;
}
```

The complete program is given in Listing 5.10.

LISTING 5.10 FutureTuition.java

```
1  public class FutureTuition {
2    public static void main(String[] args) {
3      double tuition = 10000; // Year 0
4      int year = 0;
5      while (tuition < 20000) {
6        tuition = tuition * 1.07;
7        year++;
8      }
9
10     System.out.println("Tuition will be doubled in "
11       + year + " years");
12     System.out.printf("Tuition will be $%.2f in %1d years",
13       tuition, year);
14   }
15 }
```

loop
next year's tuition

```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

The `while` loop (lines 5–8) is used to repeatedly compute the tuition for a new year. The loop terminates when the tuition is greater than or equal to `20000`.

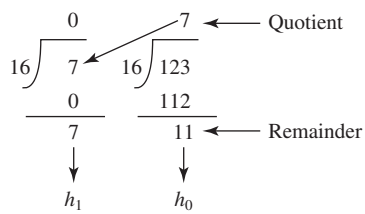
5.11.3 Case Study: Converting Decimals to Hexadecimals

Hexadecimals are often used in computer systems programming (see Appendix F for an introduction to number systems). How do you convert a decimal number to a hexadecimal number? To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits h_n, h_{n-1}, h_{n-2}, ..., h_2, h_1, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These hexadecimal digits can be found by successively dividing d by 16 until the quotient is 0. The remainders are h_0, h_1, h_2, ..., h_{n-2}, h_{n-1}, and h_n. The hexadecimal digits include the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, plus A, which is the decimal value 10; B, which is the decimal value 11; C, which is 12; D, which is 13; E, which is 14; and F, which is 15.

For example, the decimal number `123` is `7B` in hexadecimal. The conversion is done as follows. Divide `123` by `16`. The remainder is `11` (`B` in hexadecimal) and the quotient is `7`. Continue to divide `7` by `16`. The remainder is `7` and the quotient is `0`. Therefore, `7B` is the hexadecimal number for `123`.



Listing 5.11 gives a program that prompts the user to enter a decimal number and converts it into a hex number as a string.

LISTING 5.11 Dec2Hex.java

```
1  import java.util.Scanner;
2
3  public class Dec2Hex {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a decimal integer
10     System.out.print("Enter a decimal number: ");
11     int decimal = input.nextInt();                          input decimal
12
13     // Convert decimal to hex
14     String hex = "";                                        decimal to hex
15
16     while (decimal != 0) {
17       int hexValue = decimal % 16;
18
19       // Convert a decimal value to a hex digit
20       char hexDigit = (0 <= hexValue && hexValue <= 9)?      get a hex char
21         (char)(hexValue + '0'): (char)(hexValue - 10 + 'A');
22
23       hex = hexDigit + hex;                                 add to hex string
24       decimal = decimal / 16;
25     }
26
27     System.out.println("The hex number is " + hex);
28   }
29 }
```

```
Enter a decimal number: 1234 Enter
The hex number is 4D2
```

	line#	decimal	hex	hexValue	hexDigit
	14	1234	""		
iteration 1	17			2	
	23		"2"		2
	24	77			
iteration 2	17			13	
	23		"D2"		D
	24	4			
iteration 3	17			4	
	23		"4D2"		4
	24	0			

The program prompts the user to enter a decimal integer (line 11), converts it to a hex number as a string (lines 14–25), and displays the result (line 27). To convert a decimal to a hex number, the program uses a loop to successively divide the decimal number by **16** and obtain its remainder (line 17). The remainder is converted into a hex character (lines 20 and 21). The character is then appended to the hex string (line 23). The hex string is initially empty (line 14). Divide the decimal number by **16** to remove a hex digit from the number (line 24). The loop ends when the remaining decimal number becomes **0**.

The program converts a `hexValue` between `0` and `15` into a hex character. If `hexValue` is between `0` and `9`, it is converted to `(char)(hexValue +'0')` (line 21). Recall that when adding a character with an integer, the character's Unicode is used in the evaluation. For example, if `hexValue` is `5`, `(char)(hexValue + '0')` returns `5`. Similarly, if `hexValue` is between `10` and `15`, it is converted to `(char)(hexValue – 10 + 'A')` (line 21). For instance, if `hexValue` is `11`, `(char)(hexValue – 10 + 'A')` returns `B`.

5.11.1 Will the program work if `n1` and `n2` are replaced by `n1 / 2` and `n2 / 2` in line 17 in Listing 5.9?

5.11.2 In Listing 5.11, why is it wrong if you change the code `(char)(hexValue + '0')` to `hexValue + '0'` in line 21?

5.11.3 In Listing 5.11, how many times the loop body is executed for a decimal number `245`, and how many times the loop body is executed for a decimal number `3245`?

5.11.4 What is the hex number after E? What is the hex number after F?

5.11.5 Revise line 27 in Listing 5.11 so the program displays hex number 0 if the input decimal is 0.

5.12 Keywords *break* and *continue*

The `break` *and* `continue` *keywords provide additional controls in a loop.*

Pedagogical Note

Two keywords, `break` and `continue`, can be used in loop statements to provide additional controls. Using `break` and `continue` can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug. (*Note to instructors*: You may skip this section without affecting students' understanding of the rest of the book.)

You have used the keyword `break` in a `switch` statement. You can also use `break` in a loop to immediately terminate the loop. Listing 5.12 presents a program to demonstrate the effect of using `break` in a loop.

break statement

LISTING 5.12 TestBreak.java

```
1  public class TestBreak {
2    public static void main(String[] args) {
3      int sum = 0;
4      int number = 0;
5
6      while (number < 20) {
7        number++;
8        sum += number;
9        if (sum >= 100)
10         break;
11     }
12
13     System.out.println("The number is " + number);
14     System.out.println("The sum is " + sum);
15   }
16 }
```

break

```
The number is 14
The sum is 105
```

The program in Listing 5.12 adds integers from 1 to 20 in this order to sum until sum is greater than or equal to 100. Without the if statement (line 9), the program calculates the sum of the numbers from 1 to 20. However, with the if statement, the loop terminates when sum becomes greater than or equal to 100. Without the if statement, the output would be as follows:

```
The number is 20
The sum is 210
```

You can also use the continue keyword in a loop. When it is encountered, it ends the current iteration and program control goes to the end of the loop body. In other words, continue breaks out of an iteration, while the break keyword breaks out of a loop. Listing 5.13 presents a program to demonstrate the effect of using continue in a loop.

continue statement

LISTING 5.13 TestContinue.java

```
1  public class TestContinue {
2    public static void main(String[] args) {
3      int sum = 0;
4      int number = 0;
5
6      while (number < 20) {
7        number++;
8        if (number == 10 || number == 11)
9          continue;
10       sum += number;
11     }
12
13     System.out.println("The sum is " + sum);
14   }
15 }
```

continue

```
The sum is 189
```

The program in Listing 5.13 adds integers from 1 to 20 except 10 and 11 to sum. With the if statement in the program (line 8), the continue statement is executed when number becomes 10 or 11. The continue statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, number is not added to sum when it is 10 or 11. Without the if statement in the program, the output would be as follows:

```
The sum is 210
```

In this case, all of the numbers are added to sum, even when number is 10 or 11. Therefore, the result is 210, which is 21 more than it was with the if statement.

Note

The continue statement is always inside a loop. In the while and do-while loops, the loop-continuation-condition is evaluated immediately after the continue statement. In the for loop, the action-after-each-iteration is performed, then the loop-continuation-condition is evaluated immediately after the continue statement.

goto

Note

Some programming languages have a `goto` statement. The `goto` statement indiscriminately transfers control to any statement in the program and executes it. This makes your program vulnerable to errors. The `break` and `continue` statements in Java are different from `goto` statements. They operate only in a loop or a `switch` statement. The `break` statement breaks out of the loop, and the `continue` statement breaks out of the current iteration in the loop.

You can always write a program without using `break` or `continue` in a loop (see CheckPoint Question 5.12.3). In general, though, using `break` and `continue` is appropriate if it simplifies coding and makes programs easier to read.

Suppose you need to write a program to find the smallest factor other than `1` for an integer `n` (assume `n >= 2`). You can write a simple and intuitive code using the `break` statement as follows:

```
int factor = 2;
while (factor <= n) {
  if (n % factor == 0)
    break;
  factor++;
}
System.out.println("The smallest factor other than 1 for "
  + n + " is " + factor);
```

You may rewrite the code without using `break` as follows:

```
boolean found = false;
int factor = 2;
while (factor <= n && !found) {
  if (n % factor == 0)
    found = true;
  else
    factor++;
}
System.out.println("The smallest factor other than 1 for "
  + n + " is " + factor);
```

Obviously, the `break` statement makes this program simpler and easier to read in this case. However, you should use `break` and `continue` with caution. Too many `break` and `continue` statements will produce a loop with many exit points and make the program difficult to read.

Note

Programming is a creative endeavor. There are many different ways to write code. In fact, you can find a smallest factor using a rather simple code as follows:

```
int factor = 2;
while (n % factor != 0)
  factor++;
or
for (int factor = 2; n % factor != 0; factor++);
```

The code here finds the smallest factor for an integer `n`. Programming Exercise 5.16 writes a program that finds all smallest factors in `n`.

Check Point

5.12.1 What is the keyword `break` for? What is the keyword `continue` for? Will the following programs terminate? If so, give the output.

```
int balance = 10;
while (true) {
  if (balance < 9)
    break;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(a)

```
int balance = 10;
while (true) {
  if (balance < 9)
    continue;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(b)

5.12.2 The `for` loop on the left is converted into the `while` loop on the right. What is wrong? Correct it.

```
int sum = 0;
for (int i = 0; i < 4; i++) {
  if (i % 3 == 0) continue;
  sum += i;
}
```

Converted

Wrong conversion

```
int i = 0, sum = 0;
while (i < 4) {
  if (i % 3 == 0) continue;
  sum += i;
  i++;
}
```

5.12.3 Rewrite the programs `TestBreak` and `TestContinue` in Listings 5.12 and 5.13 without using `break` and `continue`.

5.12.4 After the `break` statement in (a) is executed in the following loop, which statement is executed? Show the output. After the `continue` statement in (b) is executed in the following loop, which statement is executed? Show the output.

```
for (int i = 1; i < 4; i++) {
  for (int j = 1; j < 4; j++) {
    if (i * j > 2)
      break;

    System.out.println(i * j);
  }

  System.out.println(i);
}
```

(a)

```
for (int i = 1; i < 4; i++) {
  for (int j = 1; j < 4; j++) {
    if (i * j > 2)
      continue;

    System.out.println(i * j);
  }

  System.out.println(i);
}
```

(b)

5.13 Case Study: Checking Palindromes

This section presents a program that checks whether a string is a palindrome.

A string is a palindrome if it reads the same forward and backward. The words "mom," "dad," and "noon," for instance, are all palindromes.

The problem is to write a program that prompts the user to enter a string and reports whether the string is a palindrome. One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last

think before you code

character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

Listing 5.14 gives the program.

LISTING 5.14 Palindrome.java

```
1  import java.util.Scanner;
2
3  public class Palindrome {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();
12
13     // The index of the first character in the string
14     int low = 0;
15
16     // The index of the last character in the string
17     int high = s.length() - 1;
18
19     boolean isPalindrome = true;
20     while (low < high) {
21       if (s.charAt(low) != s.charAt(high)) {
22         isPalindrome = false;
23         break;
24       }
25
26       low++;
27       high--;
28     }
29
30     if (isPalindrome)
31       System.out.println(s + " is a palindrome");
32     else
33       System.out.println(s + " is not a palindrome");
34   }
35 }
```

input string

low index

high index

update indices

```
Enter a string: noon ↵Enter
noon is a palindrome
```

```
Enter a string: abcdefgnhgfedcba ↵Enter
abcdefgnhgfedcba is not a palindrome
```

The program uses two variables, `low` and `high`, to denote the positions of the two characters at the beginning and the end in a string `s` (lines 14 and 17), as shown in the following figure.

Initially, `low` is `0` and `high` is `s.length() - 1`. If the two characters at these positions match, increment `low` by `1` and decrement `high` by `1` (lines 26–27). This process continues until (`low >= high`) or a mismatch is found (line 21).

The program uses a `boolean` variable `isPalindrome` to denote whether the string `s` is a palindrome. Initially, it is set to `true` (line 19). When a mismatch is discovered (line 21), `isPalindrome` is set to `false` (line 22) and the loop is terminated with a break statement (line 23).

5.13.1 What happens to the program if (`low < high`) in line 20 is changed to (`low <= high`)?

5.14 Case Study: Displaying Prime Numbers

This section presents a program that displays the first 50 prime numbers in 5 lines, each containing 10 numbers.

An integer greater than `1` is *prime* if its only positive divisor is `1` or itself. For example, `2`, `3`, `5`, and `7` are prime numbers, but `4`, `6`, `8`, and `9` are not.

The problem is to display the first 50 prime numbers in 5 lines, each of which contains 10 numbers. The problem can be broken into the following tasks:

- Determine whether a given number is prime.
- For `number` = `2`, `3`, `4`, `5`, `6`, …, test whether it is prime.
- Count the prime numbers.
- Display each prime number and display 10 numbers per line.

Obviously, you need to write a loop and repeatedly test whether a new `number` is prime. If the `number` is prime, increase the count by `1`. The `count` is `0` initially. When it reaches `50`, the loop terminates.

Here is the algorithm for the problem:

```
Set the number of prime numbers to be printed as
  a constant NUMBER_OF_PRIMES;
Use count to track the number of prime numbers and
  set an initial count to 0;
Set an initial number to 2;

while (count < NUMBER_OF_PRIMES) {
  Test whether number is prime;

  if number is prime {
    Display the prime number and increase the count;
  }

  Increment number by 1;
}
```

To test whether a number is prime, check whether it is divisible by `2`, `3`, `4`, and so on up to `number/2`. If a divisor is found, the number is not a prime. The algorithm can be described as follows:

```
Use a boolean variable isPrime to denote whether
  the number is prime; Set isPrime to true initially;

for (int divisor = 2; divisor <= number / 2; divisor++) {
  if (number % divisor == 0) {
    Set isPrime to false
    Exit the loop;
  }
}
```

The complete program is given in Listing 5.15.

LISTING 5.15 PrimeNumber.java

```
1  public class PrimeNumber {
2    public static void main(String[] args) {
3      final int NUMBER_OF_PRIMES = 50; // Number of primes to display
4      final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
5      int count = 0; // Count the number of prime numbers
6      int number = 2; // A number to be tested for primeness
7
8      System.out.println("The first 50 prime numbers are \n");
9
10     // Repeatedly find prime numbers
11     while (count < NUMBER_OF_PRIMES) {
12       // Assume the number is prime
13       boolean  isPrime = true; // Is the current number prime?
14
15       // Test whether number is prime
16       for (int divisor = 2; divisor <= number / 2; divisor++) {
17         if (number % divisor == 0) {  // If true, number is not prime
18           isPrime = false; // Set isPrime to false
19           break; // Exit the for loop
20         }
21       }
22
23       // Display the prime number and increase the count
24       if (isPrime) {
25         count++; // Increase the count
26
27         if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
28           // Display the number and advance to the new line
29           System.out.println(number);
30         }
31         else
32           System.out.print(number + " ");
33       }
34
35       // Check if the next number is prime
36       number++;
37     }
38   }
39 }
```

count prime numbers

check primeness

exit loop

display if prime

```
The first 50 prime numbers are
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
```

subproblem

This is a complex program for novice programmers. The key to developing a programmatic solution for this problem, and for many other problems, is to break it into subproblems and develop solutions for each of them in turn. Do not attempt to develop a complete solution in the first trial. Instead, begin by writing the code to determine whether a given number is prime, then expand the program to test whether other numbers are prime in a loop.

To determine whether a number is prime, check whether it is divisible by a number between `2` and `number/2` inclusive (lines 16–21). If so, it is not a prime number (line 18); otherwise, it is a prime number. For a prime number, display it (lines 27–33). If the count is

divisible by 10, display the number followed by a newline (lines 27–30). The program ends when the count reaches 50.

The program uses the break statement in line 19 to exit the for loop as soon as the number is found to be a nonprime. You can rewrite the loop (lines 16–21) without using the break statement, as follows:

```
for (int divisor = 2; divisor <= number / 2 && isPrime;
     divisor++) {
  // If true, the number is not prime
  if (number % divisor == 0) {
    // Set isPrime to false, if the number is not prime
    isPrime = false;
  }
}
```

However, using the break statement makes the program simpler and easier to read in this case.

Prime numbers have many applications in computer science. Section 22.7 will study several efficient algorithms for finding prime numbers.

5.14.1 Simplify the code in lines 27–32 using a conditional operator.

Key Terms

break statement 186
continue statement 187
do-while loop 171
for loop 173
infinite loop 162
input redirection 170
iteration 160
loop 160
loop body 160
nested loop 178
off-by-one error 162
output redirection 170
posttest loop 176
pretest loop 176
sentinel value 168
while loop 160

Chapter Summary

1. There are three types of repetition statements: the while loop, the do-while loop, and the for loop.

2. The part of the loop that contains the statements to be repeated is called the *loop body*.

3. A one-time execution of a loop body is referred to as an *iteration of the loop*.

4. An *infinite loop* is a loop statement that executes infinitely.

5. In designing loops, you need to consider both the *loop control structure* and the loop body.

6. The while loop checks the loop-continuation-condition first. If the condition is true, the loop body is executed; if it is false, the loop terminates.

7. The do-while loop is similar to the while loop, except the do-while loop executes the loop body first then checks the loop-continuation-condition to decide whether to continue or to terminate.

8. The while loop and the do-while loop often are used when the number of repetitions is not predetermined.

9. A *sentinel value* is a special value that signifies the end of the loop.

10. The `for` loop generally is used to execute a loop body a fixed number of times.

11. The `for` loop control has three parts. The first part is an initial action that often initializes a control variable. The second part, the `loop-continuation-condition`, determines whether the loop body is to be executed. The third part is executed after each iteration and is often used to adjust the control variable. Usually, the loop control variables are initialized and changed in the control structure.

12. The `while` loop and `for` loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.

13. The `do-while` loop is called a *posttest loop* because the condition is checked after the loop body is executed.

14. Two keywords `break` and `continue` can be used in a loop.

15. The `break` keyword immediately ends the innermost loop, which contains the break.

16. The `continue` keyword only ends the current iteration.

Quiz

Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

Programming Exercises

read and think before coding

explore solutions

Pedagogical Note

Read each problem several times until you understand it. Think how to solve the problem before starting to write code. Translate your logic into a program.

A problem often can be solved in many different ways. Students are encouraged to explore various solutions.

Sections 5.2–5.7

***5.1** (*Count positive and negative numbers and compute the average of numbers*) Write a program that reads an unspecified number of integers, determines how many positive and negative values have been read, and computes the total and average of the input values (not counting zeros). Your program ends with the input `0`. Display the average as a floating-point number. Here are sample runs:

```
Enter an integer, the input ends if it is 0: 1 2 -1 3 0 Enter
The number of positives is 3
The number of negatives is 1
The total is 5.0
The average is 1.25
```

```
Enter an integer, the input ends if it is 0: 0 Enter
No numbers are entered except 0
```

5.2 (*Repeat additions*) Listing 5.4, SubtractionQuizLoop.java, generates five random subtraction questions. Revise the program to generate 10 random addition questions for two integers between 1 and 15. Display the correct count and test time.

5.3 (*Conversion from kilograms to pounds*) Write a program that displays the following table (note 1 kilogram is 2.2 pounds):

```
Kilograms    Pounds
1              2.2
3              6.6
. . .
197          433.4
199          437.8
```

5.4 (*Conversion from miles to kilometers*) Write a program that displays the following table (note 1 mile is 1.609 kilometers):

```
Miles       Kilometers
1           1.609
2           3.218
. . .
9           14.481
10          16.090
```

5.5 (*Conversion from kilograms to pounds and pounds to kilograms*) Write a program that displays the following two tables side by side:

```
Kilograms       Pounds     |   Pounds        Kilograms
1                 2.2      |   20              9.09
3                 6.6      |   25             11.36
. . .
197             433.4      |   510           231.82
199             437.8      |   515           234.09
```

5.6 (*Conversion from miles to kilometers*) Write a program that displays the following two tables side by side:

```
Miles       Kilometers     |   Kilometers   Miles
1           1.609          |   20           12.430
2           3.218          |   25           15.538
. . .
9           14.481         |   60           37.290
10          16.090         |   65           40.398
```

****5.7** (*Financial application: compute future tuition*) Suppose the tuition for a university is $10,000 this year and increases 5% every year. In one year, the tuition will be $10,500. Write a program that displays the tuition in 10 years, and the total cost of four years' worth of tuition starting after the tenth year.

5.8 (*Find the highest score*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the name of the student with the highest score. Use the `next()` method in the `Scanner` class to read a name, rather than using the `nextLine()` method. Assume that the number of students is at least 1.

***5.9** (*Find the two highest scores*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the student with the highest score and the student with the second-highest score. Use the `next()` method in the `Scanner` class to read a name rather than using the `nextLine()` method. Assume that the number of students is at least 2.

5.10 (*Find numbers divisible by 5 and 6*) Write a program that displays all the numbers from 100 to 1,000 (10 per line) that are divisible by 5 and 6. Numbers are separated by exactly one space.

5.11 (*Find numbers divisible by 5 or 6, but not both*) Write a program that displays all the numbers from 100 to 200 (10 per line) that are divisible by 5 or 6, but not both. Numbers are separated by exactly one space.

5.12 (*Find the smallest `n` such that $n^2 > 12{,}000$*) Use a `while` loop to find the smallest integer `n` such that n^2 is greater than 12,000.

5.13 (*Find the largest `n` such that $n^3 < 12{,}000$*) Use a `while` loop to find the largest integer `n` such that n^3 is less than 12,000.

Sections 5.8–5.10

***5.14** (*Compute the greatest common divisor*) Another solution for Listing 5.9 to find the greatest common divisor of two integers `n1` and `n2` is as follows: First find `d` to be the minimum of `n1` and `n2`, then check whether `d`, `d–1`, `d–2`, …, `2`, or `1` is a divisor for both `n1` and `n2` in this order. The first such common divisor is the greatest common divisor for `n1` and `n2`. Write a program that prompts the user to enter two positive integers and displays the gcd.

***5.15** (*Display the ASCII character table*) Write a program that prints the characters in the ASCII character table from `!` to `~`. Display 10 characters per line. The ASCII table is given in Appendix B. Characters are separated by exactly one space.

***5.16** (*Find the factors of an integer*) Write a program that reads an integer and displays all its smallest factors in an increasing order. For example, if the input integer is `120`, the output should be as follows: `2`, `2`, `2`, `3`, `5`.

****5.17** (*Display pyramid*) Write a program that prompts the user to enter an integer from `1` to `15` and displays a pyramid, as presented in the following sample run:

```
Enter the number of lines: 7 Enter
                1
              2 1 2
            3 2 1 2 3
          4 3 2 1 2 3 4
        5 4 3 2 1 2 3 4 5
      6 5 4 3 2 1 2 3 4 5 6
    7 6 5 4 3 2 1 2 3 4 5 6 7
```

***5.18** (*Display four patterns using loops*) Use nested loops that display the following patterns in four separate programs:

```
Pattern A          Pattern B          Pattern C          Pattern D
1                  1 2 3 4 5 6                1          1 2 3 4 5 6
1 2                1 2 3 4 5                2 1          1 2 3 4 5
1 2 3              1 2 3 4                3 2 1            1 2 3 4
1 2 3 4            1 2 3                4 3 2 1              1 2 3
1 2 3 4 5          1 2                5 4 3 2 1                1 2
1 2 3 4 5 6        1                6 5 4 3 2 1                  1
```

**5.19 (*Display numbers in a pyramid pattern*) Write a nested `for` loop that prints the following output:

```
                    1
                  1   2   1
                1   2   4   2   1
              1   2   4   8   4   2   1
            1   2   4   8  16   8   4   2   1
          1   2   4   8  16  32  16   8   4   2   1
        1   2   4   8  16  32  64  32  16   8   4   2   1
      1   2   4   8  16  32  64 128  64  32  16   8   4   2   1
```

*5.20 (*Display prime numbers between 2 and 1,000*) Modify the program given in Listing 5.15 to display all the prime numbers between 2 and 1,000, inclusive. Display eight prime numbers per line. Numbers are separated by exactly one space.

Comprehensive

**5.21 (*Financial application: compare loans with various interest rates*) Write a program that lets the user enter the loan amount and loan period in number of years, and displays the monthly and total payments for each interest rate starting from 5% to 8%, with an increment of 1/8. Here is a sample run:

```
Loan Amount: 10000 Enter
Number of Years: 5 Enter

Interest Rate     Monthly Payment     Total Payment
5.000%            188.71              11322.74
5.125%            189.29              11357.13
5.250%            189.86              11391.59
...
7.875%            202.17              12129.97
8.000%            202.76              12165.84
```

For the formula to compute monthly payment, see Listing 2.9, ComputeLoan.java.

**5.22 (*Financial application: loan amortization schedule*) The monthly payment for a given loan pays the principal and the interest. The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principal). The principal paid for the month is therefore the monthly payment minus the monthly interest. Write a program that lets the user enter the loan amount, number of years, and interest rate then displays the amortization schedule for the loan. Here is a sample run:

VideoNote
Display loan schedule

```
Loan Amount: 10000 Enter
Number of Years: 1 Enter
Annual Interest Rate: 7 Enter

Monthly Payment: 865.26
Total Payment: 10383.21

Payment#      Interest     Principal     Balance
1                58.33        806.93      9193.07
2                53.62        811.64      8381.43
...
11               10.00        855.26       860.27
12                5.01        860.25         0.01
```

Note

The balance after the last payment may not be zero. If so, the last payment should be the normal monthly payment plus the final balance.

Hint: Write a loop to display the table. Since the monthly payment is the same for each month, it should be computed before the loop. The balance is initially the loan amount. For each iteration in the loop, compute the interest and principal, and update the balance. The loop may look as follows:

```
for (i = 1; i <= numberOfYears * 12; i++) {
  interest = monthlyInterestRate * balance;
  principal = monthlyPayment - interest;
  balance = balance - principal;
  System.out.println(i + "\t\t" + interest
    + "\t\t" + principal + "\t\t" + balance);
}
```

***5.23** (*Demonstrate cancellation errors*) A cancellation error occurs when you are manipulating a very large number with a very small number. The large number may cancel out the smaller number. For example, the result of `100000000.0 + 0.000000001` is equal to `100000000.0`. To avoid cancellation errors and obtain more accurate results, carefully select the order of computation. For example, in computing the following summation, you will obtain more accurate results by computing from right to left rather than from left to right:

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Write a program that compares the results of the summation of the preceding series, computing from left to right and from right to left with `n = 50000`.

VideoNote

Sum a series

***5.24** (*Sum a series*) Write a program to compute the following summation:

$$\frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \frac{11}{13} + \cdots + \frac{95}{97} + \frac{97}{99}$$

****5.25** (*Compute π*) You can approximate π by using the following summation:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots + \frac{(-)^{i+1}}{2i - 1}\right)$$

Write a program that displays the π value for `i = 10000`, `20000`, ..., and `100000`.

****5.26** (*Compute e*) You can approximate `e` using the following summation:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots + \frac{1}{i!}$$

Write a program that displays the `e` value for `i = 1`, `2`, ..., and `20`. Format the number to display 16 digits after the decimal point. (*Hint:* Because $i! = i \times (i - 1) \times \ldots \times 2 \times 1$, then

$$\frac{1}{i!} \text{ is } \frac{1}{i(i - 1)!}$$

Initialize `e` and `item` to be `1`, and keep adding a new `item` to `e`. The new item is the previous item divided by `i`, for `i >= 2`.)

****5.27** (*Display leap years*) Write a program that displays all the leap years, 10 per line, from 101 to 2100, separated by exactly one space. Also display the number of leap years in this period.

5.28 (*Display the first days of each month*) Write a program that prompts the user to enter the year and first day of the year, then displays the first day of each month in the year. For example, if the user entered the year **2013, and **2** for Tuesday, January 1, 2013, your program should display the following output:

```
January 1, 2013 is Tuesday
...
December 1, 2013 is Sunday
```

5.29 (*Display calendars*) Write a program that prompts the user to enter the year and first day of the year and displays the calendar table for the year on the console. For example, if the user entered the year **2013, and **2** for Tuesday, January 1, 2013, your program should display the calendar for each month in the year, as follows:

January 2013

Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

. . .

December 2013

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

*5.30 (*Financial application: compound value*) Suppose you save $100 *each* month into a savings account with the annual interest rate 5%. Thus, the monthly interest rate is **0.05 / 12 = 0.00417**. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

and so on.

Write a program that prompts the user to enter an amount (e.g., **100**), the annual interest rate (e.g., **5**), and the number of months (e.g., **6**) then displays the amount in the savings account after the given month.

***5.31** (*Financial application: compute CD value*) Suppose you put $10,000 into a CD with an annual percentage yield of 5.75%. After one month, the CD is worth

```
10000 + 10000 * 5.75 / 1200 = 10047.92
```

After two months, the CD is worth

```
10047.91 + 10047.91 * 5.75 / 1200 = 10096.06
```

After three months, the CD is worth

```
10096.06 + 10096.06 * 5.75 / 1200 = 10144.44
```

and so on.

Write a program that prompts the user to enter an amount (e.g., **10000**), the annual percentage yield (e.g., **5.75**), and the number of months (e.g., **18**) and displays a table as presented in the sample run.

```
Enter the initial deposit amount: 10000 Enter
Enter annual percentage yield: 5.75 Enter
Enter maturity period (number of months): 18 Enter

Month    CD Value
1        10047.92
2        10096.06
...
17       10846.57
18       10898.54
```

****5.32** (*Game: lottery*) Revise Listing 3.8, Lottery.java, to generate a lottery of a two-digit number. The two digits in the number are distinct. (*Hint*: Generate the first digit. Use a loop to continuously generate the second digit until it is different from the first digit.)

****5.33** (*Perfect number*) A positive integer is called a *perfect number* if it is equal to the sum of all of its positive divisors, excluding itself. For example, 6 is the first perfect number because **6 = 3 + 2 + 1**. The next is **28 = 14 + 7 + 4 + 2 + 1**. There are four perfect numbers $<$ 10,000. Write a program to find all these four numbers.

*****5.34** (*Game: scissor, rock, paper*) Programming Exercise 3.17 gives a program that plays the scissor–rock–paper game. Revise the program to let the user continuously play until either the user or the computer wins more than two times than its opponent.

***5.35** (*Summation*) Write a program to compute the following summation:

$$\frac{1}{1+\sqrt{2}}+\frac{1}{\sqrt{2}+\sqrt{3}}+\frac{1}{\sqrt{3}+\sqrt{4}}+\ \ldots\ +\frac{1}{\sqrt{624}+\sqrt{625}}$$

****5.36** (*Business application: checking ISBN*) Use loops to simplify Programming Exercise 3.9.

****5.37** (*Decimal to binary*) Write a program that prompts the user to enter a decimal integer then displays its corresponding binary value. Don't use Java's `Integer.toBinaryString(int)` in this program.

****5.38** (*Decimal to octal*) Write a program that prompts the user to enter a decimal integer and displays its corresponding octal value. Don't use Java's `Integer.toOctalString(int)` in this program.

***5.39** (*Financial application: find the sales amount*) You have just started a sales job in a department store. Your pay consists of a base salary and a commission. The base salary is $5,000. The scheme shown below is used to determine the commission rate.

Sales Amount	Commission Rate
$0.01–$5,000	8%
$5,000.01–$10,000	10%
$10,000.01 and above	12%

Note this is a graduated rate. The rate for the first $5,000 is at 8%, the next $5,000 is at 10%, and the rest is at 12%. If the sales amount is 25,000, the commission is `5,000 * 8 + 5,000 * 10 + 15,000 * 12 = 2,700`
Your goal is to earn $30,000 a year. Write a program that finds out the minimum number of sales you have to generate in order to make $30,000.

5.40 (*Simulation: heads or tails*) Write a program that simulates flipping a coin one million times and displays the number of heads and tails.

***5.41** (*Occurrence of max numbers*) Write a program that reads integers, finds the largest of them, and counts its occurrences. Assume the input ends with number **0**. Suppose you entered **3 5 2 5 5 5 0**; the program finds that the largest is **5** and the occurrence count for **5** is **4**. If no input is entered, display "No numbers are entered except 0".

(*Hint:* Maintain two variables, **max** and **count**. **max** stores the current max number and **count** stores its occurrences. Initially, assign the first number to **max** and **1** to **count**. Compare each subsequent number with **max**. If the number is greater than **max**, assign it to **max** and reset **count** to **1**. If the number is equal to **max**, increment **count** by **1**.)

```
Enter numbers: 3 5 2 5 5 5 0 Enter
The largest number is 5
The occurrence count of the largest number is 4
```

***5.42** (*Financial application: find the sales amount*) Rewrite Programming Exercise 5.39 as follows:

- Use a **for** loop instead of a **do-while** loop.
- Let the user enter **COMMISSION_SOUGHT** instead of fixing it as a constant.

***5.43** (*Math: combinations*) Write a program that displays all possible combinations for picking two numbers from integers **1** to **7**. Also display the total number of all combinations.

```
1 2
1 3
...
...

The total number of all combinations is 21
```

***5.44** (*Computer architecture: bit-level operations*) A **short** value is stored in **16** bits. Write a program that prompts the user to enter a short integer and displays the **16** bits for the integer. Here are sample runs:

```
Enter an integer: 5 Enter
The bits are 0000000000000101
```

```
Enter an integer: -5 Enter
The bits are 1111111111111011
```

(*Hint:* You need to use the bitwise right shift operator (**>>**) and the bitwise **AND** operator (**&**), which are covered in Appendix G, Bitwise Operations.)

****5.45** (*Statistics: compute mean and standard deviation*) In business applications, you are often asked to compute the mean and standard deviation of data. The mean is simply the average of the numbers. The standard deviation is a statistic that tells you how tightly all the various data are clustered around the mean in a set of data. For example, what is the average age of the students in a class? How close are the ages? If all the students are the same age, the deviation is 0.

Write a program that prompts the user to enter 10 numbers and displays the mean and standard deviations of these numbers using the following formula:

$$\text{mean} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \cdots + x_n}{n} \quad \text{deviation} = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}}$$

Here is a sample run:

```
Enter 10 numbers: 1 2 3 4.5 5.6 6 7 8 9 10 Enter
The mean is 5.61
The standard deviation is 2.99794
```

***5.46** (*Reverse a string*) Write a program that prompts the user to enter a string and displays the string in reverse order.

```
Enter a string: ABCD Enter
The reversed string is DCBA
```

***5.47** (*Business: check ISBN-13*) **ISBN-13** is a new standard for identifying books. It uses 13 digits $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}d_{11}d_{12}d_{13}$. The last digit d_{13} is a checksum, which is calculated from the other digits using the following formula:

$$10 - (d_1 + 3d_2 + d_3 + 3d_4 + d_5 + 3d_6 + d_7 + 3d_8 + d_9 + 3d_{10} + d_{11} + 3d_{12})\%10$$

If the checksum is **10**, replace it with **0**. Your program should read the input as a string. Display "invalid input" if the input is invalid. Here are sample runs:

```
Enter the first 12 digits of an ISBN-13 as a string: 978013213080 Enter
The ISBN-13 number is 9780132130806
```

```
Enter the first 12 digits of an ISBN-13 as a string: 978013213079 Enter
The ISBN-13 number is 9780132130790
```

```
Enter the first 12 digits of an ISBN-13 as a string: 97801320 Enter
97801320 is an invalid input
```

***5.48** (*Process string*) Write a program that prompts the user to enter a string and displays the characters at odd positions. Here is a sample run:

```
Enter a string: Beijing Chicago Enter
BiigCiao
```

***5.49** (*Count vowels and consonants*) Assume that the letters **A**, **E**, **I**, **O**, and **U** are vowels. Write a program that prompts the user to enter a string, and displays the number of vowels and consonants in the string.

```
Enter a string: Programming is fun Enter
The number of vowels is 5
The number of consonants is 11
```

***5.50** (*Count uppercase letters*) Write a program that prompts the user to enter a string and displays the number of the uppercase letters in the string.

```
Enter a string: Welcome to Java Enter
The number of uppercase letters is 2
```

***5.51** (*Longest common prefix*) Write a program that prompts the user to enter two strings and displays the largest common prefix of the two strings. Here are some sample runs:

```
Enter the first string: Welcome to C++ Enter
Enter the second string: Welcome to programming Enter
The common prefix is Welcome to
```

```
Enter the first string: Atlanta Enter
Enter the second string: Macon Enter
Atlanta and Macon have no common prefix
```