5.1 Introduction

Key Point

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

problem

Suppose that 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:

```
           ┌ System.out.println("Welcome to Java!");
100 times ┤  System.out.println("Welcome to Java!");
           │ ...
           └ System.out.println("Welcome to Java!");
```

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 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

Key Point

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

The syntax for the while loop is:

while loop

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

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. The

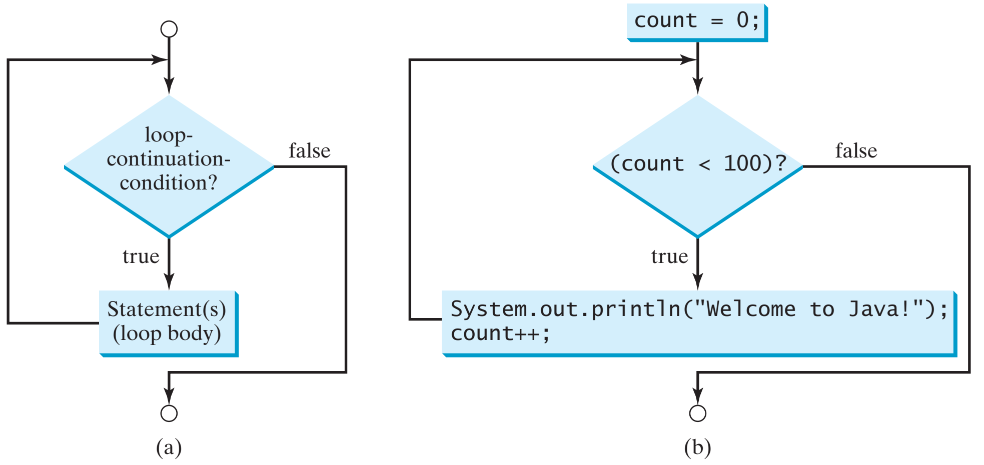


FIGURE 5.1 The `while` loop repeatedly executes the statements in the loop body when the `loop-continuation-condition` evaluates to `true`.

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

loop-continuation-condition

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

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 executions. 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`.

infinite loop

Note

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.

off-by-one error

Caution

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 shown in Listing 5.1.

Listing 5.1 RepeatAdditionQuiz.java

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

```
import java.util.Scanner;

public class RepeatAdditionQuiz {
  public static void main(String[] args) {
    int number1 = (int)(Math.random() * 10);
    int number2 = (int)(Math.random() * 10);

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    System.out.print(
      "What is " + number1 + " + " + number2 + "? ");
    int answer = input.nextInt();

    while (number1 + number2 != answer) {
      System.out.print("Wrong answer. Try again. What is "
        + number1 + " + " + number2 + "? ");
      answer = input.nextInt();
    }

    System.out.println("You got it!");
  }
}
```

```
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 Case Study: Guessing Numbers

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`**. So, 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, and 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, and then figure out how to repeatedly execute the code in a loop. For this program, you may create an initial draft, as shown in Listing 5.2.

LISTING 5.2 GuessNumberOneTime.java

```java
import java.util.Scanner;

public class GuessNumberOneTime {
  public static void main(String[] args) {
    // Generate a random number to be guessed
    int number = (int)(Math.random() * 101);

    Scanner input = new Scanner(System.in);
    System.out.println("Guess a magic number between 0 and 100");

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

    if (guess == number)
      System.out.println("Yes, the number is " + number);
```

generate a number

enter a guess

correct guess?

too high?

too low?

```
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  }
```

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. So, 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

generate a number

```
import java.util.Scanner;

public class GuessNumber {
  public static void main(String[] args) {
    // Generate a random number to be guessed
    int number = (int)(Math.random() * 101);

    Scanner input = new Scanner(System.in);
    System.out.println("Guess a magic number between 0 and 100");

    int guess = -1;
    while (guess != number) {
      // Prompt the user to guess the number
      System.out.print("\nEnter your guess: ");
```

```
15        guess = input.nextInt();
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");
21        else
22          System.out.println("Your guess is too low");
23      } // End of loop
24    }
25  }
```

enter a guess

too high?

too low?

	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.

5.2.2 Loop Design Strategies

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 like this:

```
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

5.2.3 Case Study: 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

```java
import java.util.Scanner;

public class SubtractionQuizLoop {
  public static void main(String[] args) {
    final int NUMBER_OF_QUESTIONS = 5; // Number of questions
    int correctCount = 0; // Count the number of correct answers
    int count = 0; // Count the number of questions
    long startTime = System.currentTimeMillis();
    String output = " "; // output string is initially empty
    Scanner input = new Scanner(System.in);

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

      // 2. If number1 < number2, swap number1 with number2
      if (number1 < number2) {
        int temp = number1;
        number1 = number2;
        number2 = temp;
      }

      // 3. Prompt the student to answer "What is number1 - number2?"
      System.out.print(
        "What is " + number1 + " - " + number2 + "? ");
      int answer = input.nextInt();

      // 4. Grade the answer and display the result
      if (number1 - number2 == answer) {
        System.out.println("You are correct!");
        correctCount++; // Increase the correct answer count
      }
      else
        System.out.println("Your answer is wrong.\n" + number1
          + " - " + number2 + " should be " + (number1 - number2));

      // Increase the question count
      count++;

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

get start time
loop
display a question
grade an answer
increase correct count
increase control variable
prepare output

```
43       }
44
45       long endTime = System.currentTimeMillis();
46       long testTime = endTime - startTime;
47
48       System.out.println("Correct count is " + correctCount +
49         "\nTest time is " + testTime / 1000 + " seconds\n" + output);
50     }
51   }
```

end loop

get end time

test time

display result

```
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, and computes the test time in line 46. The test time is in milliseconds and is converted to seconds in line 49.

5.2.4 Controlling a Loop with a Sentinel Value

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 writes 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

input

loop

end of loop

display result

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

5.2.5 Input and Output Redirections

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

```
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 redirection 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  output.txt
```

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

Check Point

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

5.3 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)

5.4 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);
  }
}
```

5.5 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 The `do-while` Loop

Key Point

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

The `do-while` loop is a variation of the `while` loop. Its syntax is:

`do-while` loop

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

Its execution flowchart is shown in Figure 5.2.

The loop body is executed first, and 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`

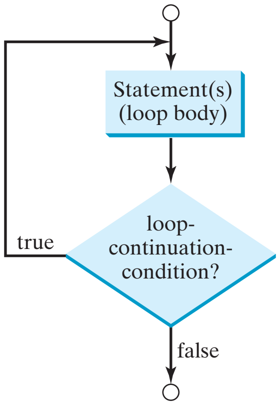


FIGURE 5.2 The `do-while` loop executes the loop body first, then checks the `loop-continuation-condition` to determine whether to continue or terminate the loop.

loop terminates. 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 executed. 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 shown in Listing 5.6.

LISTING 5.6 TestDoWhile.java

```java
import java.util.Scanner;

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

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Keep reading data until the input is 0
    do {                                              // loop
      // Read the next data
      System.out.print(
        "Enter an integer (the input ends if it is 0): ");
      data = input.nextInt();

      sum += data;
    } while (data != 0);                              // end loop

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

```
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.

Check Point

5.6 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);
  }
}
```

5.7 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();
}
```

5.4 The for Loop

Key Point

*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
}
```

A `for` loop can be used to simplify the preceding loop as:

```
for (i = initialValue; i < endValue; i++)
  // Loop body
  ...
}
```

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

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.

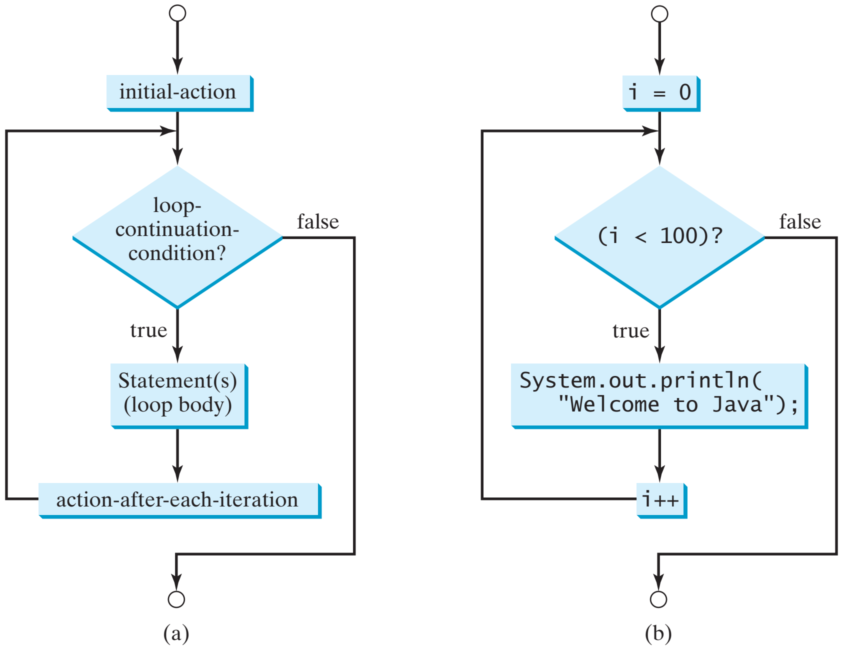


FIGURE 5.3 A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the `loop-continuation-condition` evaluates to `true`.

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

A `for` loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a *control variable*. The `initial-action` often initializes a control variable, the `action-after-each-iteration` usually increments or decrements the control variable, and the `loop-continuation-condition`

control variable

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).

(a)		(b)		(c)

`for ( ; ; ) {`
`  // Do something`
`}`
	Equivalent	`for ( ; true; ) {`
		`  // Do something`
		`}`
			Equivalent
			This is better
				`while (true) {`
				`  // Do something`
				`}`

Check Point

5.8 Do the following two loops result in the same value in `sum`?

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

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

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

5.10 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);
  }
}
```

5.11 What does the following statement do?

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

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

5.13 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;
```

5.14 Count the number of iterations in the following loops.

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

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

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

(c)

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

(d)

5.5 Which Loop to Use?

Key Point

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

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 5.25 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.

Error

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

(a)

Empty body

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

(b)

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

Error

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

(c)

Empty body

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

(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);
```

Correct

Check Point

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

5.16 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++;
}
```

5.17 Identify and fix the errors in the following code:

```
public class Test {
  public void main(String[] args) {
    for (int i = 0; i < 10; i++);
      sum += i;

    if (i < j);
      System.out.println(i)
    else
      System.out.println(j);

    while (j < 10);
    {
      j++;
    }

    do {
      j++;
    } while (j < 10)
  }
}
```

5.18 What is wrong with the following programs?

```java
public class ShowErrors {
  public static void main(String[] args) {
    int i = 0;
    do {
      System.out.println(i + 4);
      i++;
    }
    while (i < 10)
  }
}
```

(a)

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

(b)

5.6 Nested Loops

A loop can be nested inside another loop.

nested 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.

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

LISTING 5.7 MultiplicationTable.java

table title

outer loop

inner loop

```java
public class MultiplicationTable {
  /** Main method */
  public static void main(String[] args) {
    // Display the table heading
    System.out.println("           Multiplication Table");

    // Display the number title
    System.out.print("    ");
    for (int j = 1; j <= 9; j++)
      System.out.print("   " + j);

    System.out.println("\n-----------------------------------------");

    // Display table body
    for (int i = 1; i <= 9; i++) {
      System.out.print(i + " | ");
      for (int j = 1; j <= 9; j++) {
        // Display the product and align properly
        System.out.printf("%4d", i * j);
      }
      System.out.println();
    }
  }
}
```

```
           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 9–10) displays the numbers `1` through `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 that 1 microsecond is one millionth (10^{-6}) of a second.

Check Point

5.19 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)
```

5.20 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)

5.7 Minimizing Numeric Errors

Key Point

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

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 7–8) 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 that 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 that 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 that the floating-point numbers are represented by approximation. To fix the problem, use an integer count to ensure that 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) 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 is one way to minimize errors.

5.8 Case Studies

Key Point

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 four additional examples of solving problems using loops.

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

think before you 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. So, 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. Once you have a logical solution, type the code to translate the solution into a Java program as follows:

logical solution

```
  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

input
input
gcd
check divisor
output

```java
import java.util.Scanner;

public class GreatestCommonDivisor {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter first integer: ");
    int n1 = input.nextInt();
    System.out.print("Enter second integer: ");
    int n2 = input.nextInt();

    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++;
    }

    System.out.println("The greatest common divisor for " + n1 +
      " and " + n2 + " is " + gcd);
  }
}
```

```
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:

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

multiple solutions

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).

erroneous 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:

```java
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 5.21 for the answer.

5.8.2 Case Study: Predicting the Future Tuition

Suppose that 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 shown 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**.

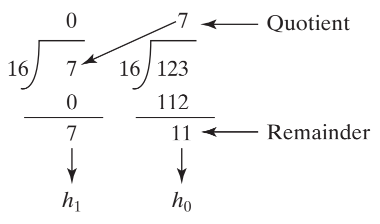
5.8.3 Case Study: Converting Decimals to Hexadecimals

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 $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1$, 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 $h_0, h_1, h_2, \ldots, h_{n-2}, h_{n-1}$, 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 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`

input decimal

decimal to hex

```
import java.util.Scanner;

public class Dec2Hex {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a decimal integer
    System.out.print("Enter a decimal number: ");
    int decimal = input.nextInt();

    // Convert decimal to hex
    String hex = "";

    while (decimal != 0) {
      int hexValue = decimal % 16;

      // Convert a decimal value to a hex digit
      char hexDigit = (hexValue <= 9 && hexValue >= 0) ?
        (char)(hexValue + '0') : (char)(hexValue - 10 + 'A');

      hex = hexDigit + hex;
```

```
24          decimal = decimal / 16;
25        }
26
27        System.out.println("The hex number is " + hex);
28      }
29    }
```

get a hex char

get a letter

```
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 20–21). 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**.

Check Point

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

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

5.23 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**?

5.9 Keywords *break* and *continue*

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

Key Point

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.)

break statement

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.

Listing 5.12 TestBreak.java

break

```
 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 }
```

```
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**. But with the **if** statement, the loop terminates when **sum** becomes greater than or equal to **100**. Without the **if** statement, the output would be:

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

continue statement

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.

Listing 5.13 TestContinue.java

```
public class TestContinue {
  public static void main(String[] args) {
    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.

You can always write a program without using `break` or `continue` in a loop (see Checkpoint Question 5.26). 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)
```

```java
      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.

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.

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:

```java
int factor = 2;
while (factor <= n && n % factor != 0)
  factor++;
```

Check Point

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

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

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

(a)

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

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

(b)

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

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

Converted

Wrong conversion

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

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

5.27 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)

5.10 Case Study: Checking Palindromes

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

Key Point

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 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.

think before you code

Listing 5.14 gives the program.

LISTING 5.14 Palindrome.java

```
import java.util.Scanner;

public class Palindrome {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    // The index of the first character in the string
    int low = 0;

    // The index of the last character in the string
    int high = s.length() - 1;

    boolean isPalindrome = true;
    while (low < high) {
      if (s.charAt(low) != s.charAt(high)) {
        isPalindrome = false;
        break;
      }

```

input string

low index

high index

update indices

```
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   }
```

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

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

The program uses two variables, **low** and **high**, to denote the position of the two characters at the beginning and the end in a string **s** (lines 14, 17). 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 palindrome. Initially, it is set to **true** (line 19). When a mismatch is discovered (line 21), **isPalindrome** is to **false** (line 22) and the loop is terminated with a break statement (line 23).

5.11 Case Study: Displaying Prime Numbers

Key Point

This section presents a program that displays the first fifty prime numbers in five lines, each containing ten 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 five lines, each of which contains ten 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 ten 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

```java
public class PrimeNumber {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 50; // Number of primes to display
    final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    System.out.println("The first 50 prime numbers are \n");

    // Repeatedly find prime numbers
    while (count < NUMBER_OF_PRIMES) {
      // Assume the number is prime
      boolean isPrime = true; // Is the current number prime?

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

      // Display the prime number and increase the count
      if (isPrime) {
        count++; // Increase the count

        if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
          // Display the number and advance to the new line
          System.out.println(number);
        }
        else
          System.out.print(number + " ");
```

count prime numbers

check primeness

exit loop

display if prime

```
33        }
34
35        // Check if the next number is prime
36        number++;
37      }
38    }
39  }
```

```
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. If the count is divisible by **10** (lines 27–30), advance to a new line. 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.

Key Terms