Chapter 5

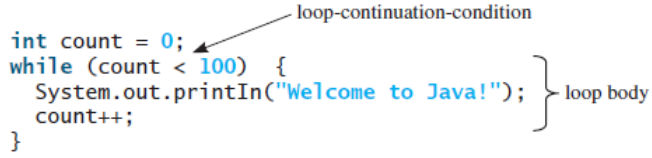
Loops

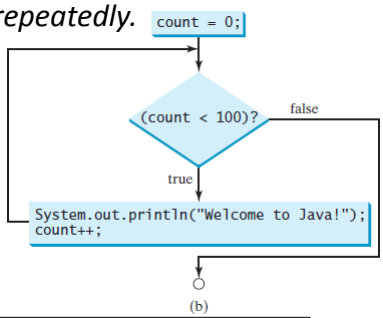
Objectives

- To write programs for executing statements repeatedly using a **while** loop
- To control a loop with a sentinel value
- To obtain large input from a file using input redirection rather than typing from the keyboard
- To write loops using **do-while** statements
- To write loops using **for** statements
- To discover the similarities and differences of three types of loop statements
- To write nested loops
- To implement program control with **break** and **continue**

Loops: an introduction

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





- 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.
- 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.*

Loops: an introduction (cont'd)

- *Loops* are constructs that control repeated executions of a block of statements. Java provides three types of loop statements:
  - **while** loops,
  - **do**-**while** loops, and
  - **for** loops.

- **while** loops
- **do-while** loops
- **for** loops

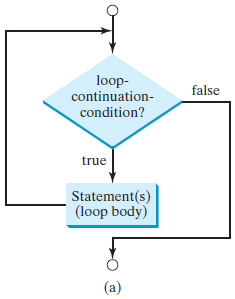
The while loop

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

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

- 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.

- 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.



- **while** loops
- **do-while** loops
- **for** loops

The while loop: an example

- 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,

LISTING 3.1 AdditionQuiz.java

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

LISTING 5.1 RepeatAdditionQuiz.java

```
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 }
```

A case study on guessing numbers

- 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
```


A case study on guessing numbers (cont'd)

- 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.
- 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

```
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 }
```

A case study on guessing numbers (cont'd)

- When you run Listing 5.2, 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 (guess != number) {
  // lines 11-20 here
}
```

- This loop repeatedly prompts the user to enter a guess until **guess** matches **number**, then the loop should end.
- There is a *compile error* after adding the above loop, how to fix it?

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 }
```


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;
}
```

- Rewrite Listing 3.3, SubtractionQuiz.java so that it generates 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** SubtractionQuizLoop.java is a solution to this requirement.

Controlling a Loop with a Sentinel Value

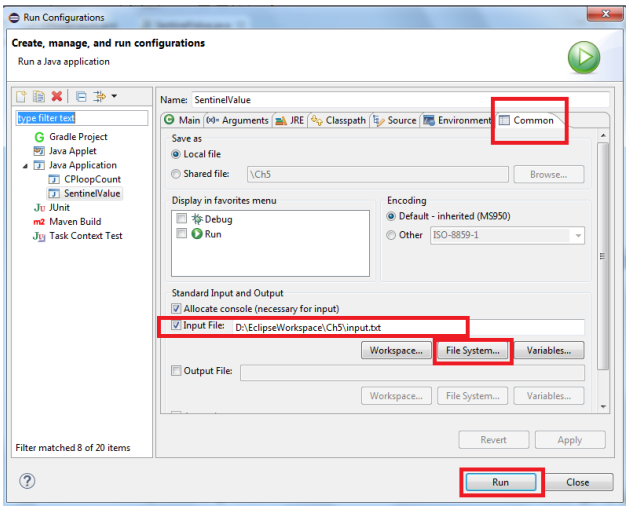
- Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a *sentinel value*, signifies the end of the input.
- A loop that uses a sentinel value to control its execution is called a *sentinel-controlled 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.
- Note that if the first input read is **0**, the loop body never executes, and the resulting sum is **0**.

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(
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 }
```


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 create an input file from eclipse directly via File → New → Untitled Text File.
- You can store the data separated by whitespaces in a text file, say **input.txt.** Remember to add a newline to the end of your input file.
- Open Run → *Run Configurations..* window. At the configuration tabs, go to *Common* tab. There you can specify an input file.
- The program then takes the input from the file **input .txt** rather than having the user type the data from the keyboard at runtime.
- Similarly, there is *output redirection*, which sends the output to a file rather than displaying it on the console.

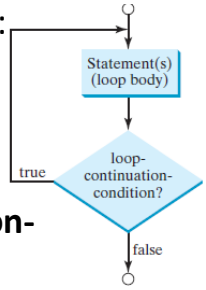


- **while** loops
- **do-while** loops
- **for** loops

The do-while Loop

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



- The loop body is executed first, and then the **loop-continuation-condition** is evaluated.
- Use a **do-while** loop if you have statements inside the loop that must be executed *at least once*.

- **while** loops
- **do-while** loops
- **for** loops

The do-while Loop: an example

- Rewrite the **while** loop in Listing 5.5 using a **do-while** loop, as shown in Listing 5.6. Which one is a more convenient choice and why?

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(
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 }
```

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 {
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);
21
22     System.out.println("The sum is " + sum);
23   }
24 }
```

- **while** loops
- **do**-**while** loops
- **for** loops

The for Loop

- *A* **for** *loop has a concise syntax for writing loops.* The syntax is:

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

- 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.

- **while** loops
- **do**-**while** loops
- **for** loops

The for Loop

- Often you write a loop in the following common form:

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

equivalent to

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

- The control variable (that is variable i as in the code above) 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. For example: **for** (**int** i = **0**; i < **100**; i++) {…}
- If the variable is declared inside the loop control structure, it cannot be referenced outside the loop.

- **while** loops
- **do-while** loops
- **for** loops

The for Loop: flowchart

- The flowchart of the **for** loop is shown in Figure 5.3a
- Write the code for the flowchart in Figure 5.3b and write a sentence to state what the for loop does.

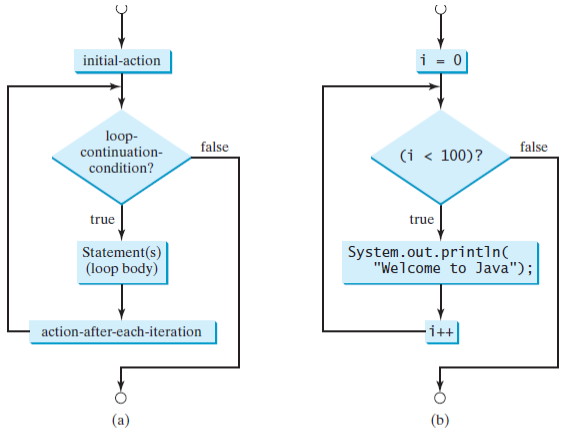


FIGURE 5.3 A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the `loop-continuation-condition` evaluates to `true`.

- **while** loops
- **do-while** loops
- **for** loops

The for Loop: some additional notes

- 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:

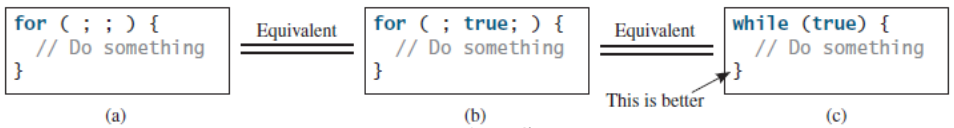
```
for (int i = 0, j = 0; i + j < 10; i++, j++) {
    // Do something
}
```

- 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++);
```

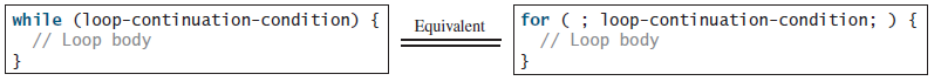
  - This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action and increment or decrement the control variable as an action after each iteration.

- 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).



Which loop to use?

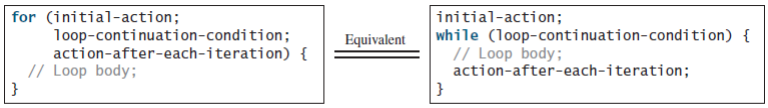
- *You can use a* **for** *loop, a* **while** *loop, or a* **do-while** *loop, whichever is convenient.*
- The **while** loop and **for** loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.
- The **do-while** loop is called a *posttest loop* 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).



(a) (b)

Which loop to use? (cont'd)

- A **for** loop in (a) in the following figure can generally be converted into the **while** loop in (b) except in certain special cases



(a) (b)

- 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.

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

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);
11
12     System.out.println("\n--------------------------------------");
13
14     // Display table body
15     for (int i = 1; i <= 9; i++) {
16       System.out.print(i + " | ");
17       for (int j = 1; j <= 9; j++) {
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
```


Keyword *break*

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

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 }
```

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

Keyword *continue*

- 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.
- The program in Listing 5.13 adds integers from **1** to **20** except **10** and **11** to **sum**.
- 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**.

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 }
```

```
The sum is 189
```


Keyword *break:* with or without

- You can always write a program without using **break** or **continue** in a loop.
- Suppose you need to write a program to find the smallest factor other than **1** for an integer **n** (assume **n >= 2**).

```
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);
```

```
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);
```

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:

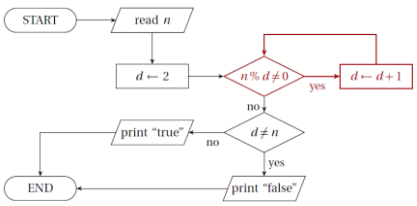
```
int factor = 2;
while (factor <= n && n % factor != 0)
  factor++;
```

Case Study: Displaying Prime Numbers

- Let's see how *to display the first fifty prime numbers in five lines, each containing ten numbers.*
- 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 sub-problems 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.
- The problem can be broken into the following tasks:
  - 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.

Case Study: Displaying Prime Numbers (cont'd)

- 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.
- 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;
  }
}
```

Case Study: Displaying Prime Numbers (cont'd)

- 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.

Case Study: Displaying Prime Numbers – Coding

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;
}
```

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) {                          count prime numbers
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++) {  check primeness
17         if (number % divisor == 0) { // If true, number is not prime
18           isPrime = false; // Set isPrime to false
19           break; // Exit the for loop                           exit loop
20         }
21       }
22
23       // Display the prime number and increase the count
24       if (isPrime) {                                            display if prime
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 }
```

Case Study: Displaying Prime Numbers – another solution

- 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;
  }
}
```

- The output of Listing 5.15 is as follows:

```
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
```


Chapter Summary

- There are three types of repetition statements: the **while** loop, the **do-while** loop, and the **for** loop.
- 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 of the loop.*
- An *infinite loop* is a loop statement that executes infinitely.
- In designing loops, you need to consider both the *loop control structure* and the loop body.
- 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.
- The **do-while** loop is similar to the **while** loop, except that the **do-while** loop executes the loop body first and then checks the **loop-continuation-condition** to decide whether to continue or to terminate.

Chapter Summary

- The **while** loop and the **do-while** loop often are used when the number of repetitions is not predetermined.
- A *sentinel value* is a special value that signifies the end of the loop.
- The **for** loop generally is used to execute a loop body a fixed number of times.
- 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.
- Two keywords, **break** and **continue**, can be used in a loop.
- The **break** keyword immediately ends the innermost loop, which contains the break.
- The **continue** keyword only ends the current iteration.

Ideas for further practice

- How many times is the **println** statement executed? Set a breakpoint and run the code in debug mode to step-over to study the changes in the values of the variables.
- Set a breakpoint to study the changes in the values of the variables for these two sets of code as well.
- Read 5.8.1 Case Study: Finding the Greatest Common Divisor
- Read 5.10 Case Study: Checking Palindromes

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

```
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)

Ideas for further practice

- 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)

33