For-Loops

7

7.1 Using For-Loops for Repetition

A **loop** is a program structure for executing a block of code repeatedly. The code for a loop usually comes with a specification of when to stop the repetition. Java has four loop structures: the for-loop, the while-loop, the do-while loop, and the for-each loop. We study the for-loop in this chapter, the while-loop and do-while loop in Chap. 11, and the for-each loop in Chaps. 17 and 18.

In Sect. 5.2, we studied computing the BMI value on height and weight values that the user enters. In `BMIInteractive.java` (List 7.1) we used a method named `oneInteraction` that received one pair of height and weight from the user and reported the BMI value with respect to the combination. The action of the method `main` in the program was to call `oneInteraction` twice.

Suppose we want to increase the number of repetitions from two to a much larger number, for example, ten. We can accomplish this by adding eight more lines of `oneInteraction();`:

```
1   public static void main( String[] args )
2   {
3     oneInteraction();
4     oneInteraction();
5     oneInteraction();
6     oneInteraction();
7     oneInteraction();
8     oneInteraction();
9     oneInteraction();
10    oneInteraction();
11    oneInteraction();
12    oneInteraction();
13  }
```

Now what should we do if we wanted to increase the number of repetitions to 20? Should we add ten more lines of the same `oneInteraction();`? Using a loop, it is possible to state the 20-time repetitions in just a few lines.

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_7

```
1  public static void main( String[] args )
2  {
3    int i;
4    for ( i = 1; i <= 10; i = i + 1 )
5    {
6      oneInteraction();
7    }
8  }
```

Line 4 of the code,

```
for ( i = 1; i <= 10; i = i + 1 )
```

is the for-loop. It means

"repeat the following as long as `i <= 10` by first assigning the value of 1 to `i` and then adding 1 to `i` each time."

The "following" refers to the block of code between Lines 5–7. We call this block the **loop-body**.

The actions that take place in the above for-loop are as follows:

- The value of `1` is stored in `i`.
- As long as the value of `i` is less than or equal to `10`,
  - execute `oneInteraction` and
  - increase the value of `i` by 1.

The use of a for-loop in stating the repetition makes it easy to change the number of repetitions. Furthermore, if the name of the method changes, we only have to replace just one call, which appears in the body of the loop.

The variable `i` that refers to the "round" in the repetition can be used in the body of the loop. For example, before calling `oneInteraction` we can announce the round:

```
1  public static void main( String[] args )
2  {
3    int i;
4    for ( i = 1; i <= 10; i = i + 1 )
5    {
6      System.out.println( "This is round " + i + "." );
7      oneInteraction();
8    }
9  }
```

The code with the round announcement appears next:

```
1  import java.util.Scanner;
2  public class BMIRepetitive
3  {
4    public static final double BMI_SCALE = 703.0;
5    public static final int FEET_TO_INCHES = 12;
6
7    public static double bmiFormula( double weight, double height )
8    {
9      return BMI_SCALE * weight / (height * height);
10   }
11
12   public static void oneInteraction()
13   {
14     Scanner keyboard = new Scanner( System.in );
15     System.out.print( "Enter weight: " );
16     double weight = keyboard.nextDouble();
17     System.out.print( "Enter height in feet and inches: " );
18     double feet = keyboard.nextDouble();
19     double inches = keyboard.nextDouble();
20     double height = FEET_TO_INCHES * feet + inches;
21     double bmi = bmiFormula( weight, height );
22     System.out.println( "Your BMI is " + bmi + "." );
23   }
24   public static void main( String[] args )
25   {
26     int i;
27     for ( i = 1; i <= 10; i = i + 1 )
28     {
29       System.out.println( "This is round " + i + "." );
30       oneInteraction();
31     }
32   }
33 }
```

Listing 7.1 A program that repeatedly computes BMI using a for-loop. The program announces each round

The execution of the code, with some input from the user, produces the following:

```
1  This is round 1.
2  Enter weight: 150
3  Enter height in feet and inches: 5 6
4  Your BMI is 24.207988980716255.
5  This is round 2.
6  Enter weight: 150 5 7
7  Enter height in feet and inches: Your BMI is 23.490755179327245.
8  This is round 3.
9  Enter weight: 160 5 7
10 Enter height in feet and inches: Your BMI is 25.056805524615726.
11 This is round 4.
12 ...
```

As we have seen in the above, the header part of a for-loop has three components with a semicolon in between:

- **initialization**,
- **continuation (termination) condition**,
- **update**

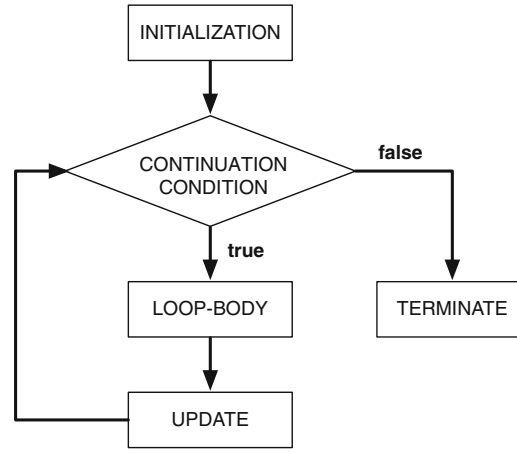

Fig. 7.1 A generic flow chart of for-loops

In other words, the header part of a for-loop takes the form of:

```
for ( INITIALIZATION; CONTINUATION CONDITION; UPDATE ) { ... }
```

The roles of these components are as follows:

- The initialization of a for-loop is a statement that is executed prior to entering the repetition.
- The continuation condition is one that must hold for the loop-body to execute. Before executing the loop-body, this condition is tested. If the condition does not hold, the loop is terminated immediately.
- The update is a statement that is executed after each execution of the loop-body.[1]

The roles of the three components are summarized in Fig. 7.1. Most typically, in a for-loop, the initialization is an assignment to a variable (usually an integer), the termination condition is a comparison involving the variable, and the update is a modification to the variable. We call such a variable an **iteration variable**.

The next program contains a for-loop with an interaction that changes its value from 0 to 7.

```
1  public class ForExample
2  {
3    public static void main( String[] args )
4    {
5      int count;
6      for ( count = 0; count < 8; count ++ )
7      {
8        System.out.println( "The value of count is " + count );
9      }
10   }
11 }
```

Listing 7.2 An iteration over the sequence `0, ..., 7`

[1] Both the initialization and the update can be composed of multiple statements. Those statements must be separated by a comma in between.

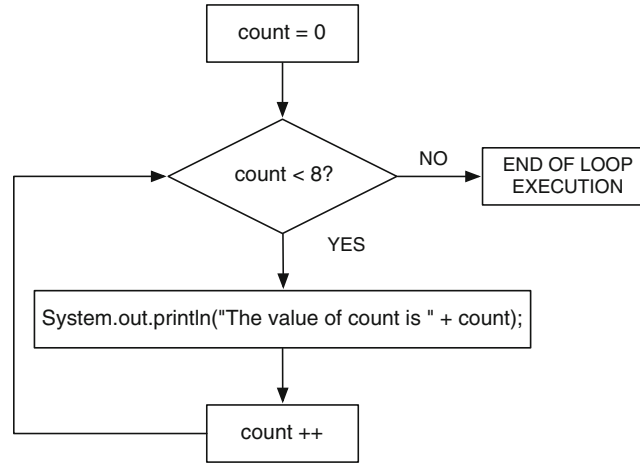

Fig. 7.2 The code execution diagram of `ForExample`

The iteration variable of the for-loop is `count`. Since the initial value of the iteration variable is 0, the termination condition is `count < 8`, and the update is `count ++`, the last value of `count` for which the loop-body is executed is 7. Figure 7.2 summarizes the above observation.

Running the code produces the following:

```
1 The value of count is 0
2 The value of count is 1
3 The value of count is 2
4 The value of count is 3
5 The value of count is 4
6 The value of count is 5
7 The value of count is 6
8 The value of count is 7
```

Consider changing the initialization to:

```
count = 8
```

while retaining the other two components of the loop header.

```
1  public class ForExampleNoOutput
2  {
3    public static void main( String[] args )
4    {
5      int count;
6      for ( count = 8; count < 8; count ++ )
7      {
8        System.out.println( "The value of count is " + count );
9      }
10   }
11 }
```

Listing 7.3 A for-loop that never executes its body

Since the new initial value already fails to satisfy the termination condition, the loop terminates without executing its body once.

Consider further changing the comparison in the termination condition to `count <= 8`, as shown next:

```
1  public class ForExampleJustOnce
2  {
3    public static void main( String[] args )
4    {
5      int count;
6      for ( count = 8; count <=8; count ++ )
7      {
8        System.out.println( "The value of count is " + count );
9      }
10   }
11 }
```

Listing 7.4 A for-loop that never executes its body

The loop-body will then execute just once, producing the output line:

```
The value of count is 8
```

Remember that the existence/absence of the equal sign often makes a significant difference in the behavior of a for-loop.

The variable type declaration of the iteration variable can be inserted in the initialization, like this:

```
1    public static void main( String[] args )
2    {
3      // int count;
4      for ( int count = 0; count < 8; count ++ )
5      {
6        System.out.println( "The value of count is " + count );
7      }
8    }
```

This requires the removal of the declaration of the variable `count` appearing in Line 3; otherwise, the scopes of the two declarations of the identical variable names overlap. Retaining the declaration (i.e., not adding the two forward slashes) will result in an error message during compilation, e.g.:

```
1  ForExample.java:6: error: variable count is already defined in method
       main(String[])
2      for ( int count = 0; count < 8; count ++ )
3                ^
4  1 error
```

Here, the code uses `//` to make the line a comment.

If the iteration variable is declared in the initialization of a for-loop, then its scope is the entire for-loop, including the header-part. The scope of the variable `count` becomes the entire for-loop, not the entire `main`.

Let us go back to the use of a for-loop to repeat an action.

Suppose we want to generate a random number between 0 and 1 (using `Math.random`) N times and take the average of the random numbers. What can we expect? To compute the average, we only have to compute the total of the random numbers and then divide it by N. According to the statistical principal called the **law of large numbers**, if the random number generation is fair, the larger the value of N is, the closer the average is to 0.50.

Suppose we use a modest value, 20, for N and run an experiment to examine how close the average gets to the expected average of 0.50. Our program uses a `double` variable named `random` to record the random number `Math.random` produces at each round, an `int` variable named `round` to record the number of times a random number has been generated, and a `double` variable named `sum` to record the sum of the random numbers. At each round, the program announces the round number and the random number that has been generated. After the for-loop, the program divides the total by 20 and then prints the average.

Here is the code that follows this algorithm:

```
1  public class RandomAverage
2  {
3    public static void main( String[] args )
4    {
5      double random, sum;
6      sum = 0;
7      for ( int round = 1; round <= 20; round ++ )
8      {
9        random = Math.random();
10       sum += random;
11       System.out.print( "round=" + round );
12       System.out.println( ", number=" + random );
13     }
14     sum /= 20;
15     System.out.println( "average=" + sum );
16   }
17 }
```

Listing 7.5 A program that computes the average of 20 random numbers

Here is the result of one execution. Since the program uses a random number generator, it is unlikely that the same output result occurs twice:

```
1  round=1, number=0.8157853753717103
2  round=2, number=0.46775040606117546
3  round=3, number=0.8441866465531849
4  round=4, number=0.5829690931048322
5  round=5, number=0.5599437446060029
6  round=6, number=0.8672105406302983
7  round=7, number=0.33033589637735683
8  round=8, number=0.5881510375862207
9  round=9, number=0.38566559572527037
10 round=10, number=0.31425625823536696
11 round=11, number=0.3286394398265723
12 round=12, number=0.9437308791253096
13 round=13, number=0.8607491030785093
14 round=14, number=0.008072130126899335
15 round=15, number=0.0477975147654055
16 round=16, number=0.2659779860979148
17 round=17, number=0.5987723989794204
18 round=18, number=0.8369910748406596
19 round=19, number=0.25924195790278215
20 round=20, number=0.005168580596320194
21 average=0.4955697829795607
```

Here is another run:

```
java RandomAverage
round=1, number=0.8920844233690451
round=2, number=0.3288527172059581
round=3, number=0.4933769583576755
round=4, number=0.8788336888081347
round=5, number=0.6212225634348153
round=6, number=0.7371533245256305
round=7, number=0.7485906839518522
round=8, number=0.5634098385103024
round=9, number=0.22549810357332067
round=10, number=0.8546489284189476
round=11, number=0.2905654125567769
round=12, number=0.24907615852772835
round=13, number=0.5800520831995266
round=14, number=0.2043575262026217
round=15, number=0.25741759539019937
round=16, number=0.4710648380419118
round=17, number=0.04850748397632787
round=18, number=0.6508157292281656
round=19, number=0.33074890481577923
round=20, number=0.7833542508320545
average=0.5104815606463388
```

In both cases, the average is quite close to 0.50.

7.2 Iteration

7.2.1 Simple Iteration

We formally call the process of generating a series of values using a loop an **iteration**. For-loops are useful for iteration. In many cases of iterations with a for-loop, the series is generated as the values of its iteration variable.

Using different initializations, termination conditions, and updating actions, it is possible to produce a wide variety of sequences. Here are some examples:

```
    for ( int count = 10; count >= 1; count -- );
    for ( int count = 10; count > 0; count -= 2 );
    for ( int count = 89; count <= 100; count += 3 );
    for ( int count = 1; count <= 100; count *= 2 );
    for ( int count = 3; count <= 80; count = count * 2 - 1 );
```

These are all syntactically correct. Their iteration variables are named `count`. Each line ends with a semicolon, which indicates that there is no loop-body. The learners of Java often attach a semicolon after the closing parenthesis, which may produce a source code that compiles but does not run as anticipated.

The sequences generated by the for-loops are:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
10, 8, 6, 4, 2
89, 92, 95, 98
1, 2, 4, 8, 16, 32, 64
3, 7, 15, 31, 63
```

The next code executes the for-loop (now without the semicolon after the closing parenthesis) and prints the values generated in one single line for each loop, with a single white space character inserted before each value. To do this, the code uses `print` instead of `println`. Because `print` does not add the newline character at the end of its loop, the code must execute one `println` statement to go to the next line.

Furthermore, the two lines preceding each for-loop have the role of producing the three components of the loop on the screen:

```
1  public class IterationSamples
2  {
3    public static void main( String[] args )
4    {
5      int count;
6      System.out.println( "----- Loop Number 1 ----- " );
7      System.out.println( "count = 10; count >= 1; count --" );
8      for ( count = 10; count >= 1; count -- )
9      {
10       System.out.print( " " + count );
11     }
12     System.out.println( "\n----- Loop Number 2 ----- " );
13     System.out.println( "count = 10; count > 0; count -= 2" );
14     for ( count = 10; count > 0; count -= 2 )
15     {
16       System.out.print( " " + count );
17     }
```

Listing 7.6 Examples of iteration (part 1)

```
18     System.out.println( "\n----- Loop Number 3 ----- " );
19     System.out.println( "count = 89; count <= 100; count += 3" );
20     for ( count = 89; count <= 100; count += 3 )
21     {
22       System.out.print( " " + count );
23     }
24     System.out.println( "\n----- Loop Number 4 ----- " );
25     System.out.println( "count = 1; count <= 100; count *= 2" );
26     for ( count = 1; count <= 100; count *= 2 )
27     {
28       System.out.print( " " + count );
29     }
30     System.out.println( "\n----- Loop Number 5 ----- " );
31     System.out.println(
32          "count = 3; count <= 80; count = count * 2 - 1" );
33     for ( count = 3; count <= 80; count = count * 2 - 1 )
34     {
35       System.out.print( " " + count );
36     }
37     System.out.println();
38   }
39 }
```

Listing 7.7 Examples of iteration (part 2)

The code produces the following output:

```
----- Loop Number 1 -----
count = 10; count >= 1; count --
 10 9 8 7 6 5 4 3 2 1
----- Loop Number 2 -----
count = 10; count > 0; count -= 2
 10 8 6 4 2
----- Loop Number 3 -----
count = 89; count <= 100; count += 3
 89 92 95 95 98
----- Loop Number 4 -----
count = 1; count <= 100; count *= 2
 1 2 4 8 16 32 64
----- Loop Number 5 -----
count = 3; count <= 80; count = count * 2 - 1
 3 5 9 17 33 65
```

Using an iteration that decreases the value of its iteration variable from its start value, as well as a `Scanner` object that receives the start value, we can write the following simple program for executing a countdown from the start number to 0.

Note the following specifics of the code:

- The program uses a `Scanner` object. Line 1 is an `import` statement for using a `Scanner`. We use a generic `java.util.*` here.
- In Line 9, the program receives the start value and stores it in a variable named `start`. The variable `start` is declared in Line 6.
- The iterator has `value > 0` as the termination condition. Thus, the last value of `value` in which the loop-body executes is greater than 0. Since `value` is an integer, the smallest integer that satisfies `value > 0` is 1, so the condition is equivalent to `value >= 1`.

```
1  import java.util.* ;
2
3  public class CountDown
4  {
5    public static void main( String[] args )
6    {
7      Scanner keyboard;
8      int start, value;
9      keyboard = new Scanner( System.in );
10     System.out.print( "Enter start: " );
11     start = keyboard.nextInt();
12     for ( value = start; value > 0; value -- )
13     {
14       System.out.println( ".." + value );
15     }
16     System.out.println( "BOOOOOOOOOOOOOOOOOOOOOOOOM!" );
17   }
18 }
```

Listing 7.8 A program that counts down from an input number to 0

Executing the code produces the following result:

```
Enter start: 10
..10
..9
..8
..7
..6
..5
..4
..3
..2
..1
BOOOOOOOOOOOOOOOOOOOOOOOOM!
```

Another run produces this result:

```
Enter start: 13
..13
..12
..11
..10
..9
..8
..7
..6
..5
..4
..3
..2
..1
BOOOOOOOOOOOOOOOOOOOOOOOOM!
```

We can accelerate the speed at which the number decreases by using "dividing by 2" instead of "decreasing by 1" in the iteration, a shown next:

```
import java.util. * ;

public class CountDownFast
{
  public static void main( String[] args )
  {
    Scanner keyboard;
    int start, value;
    keyboard = new Scanner( System.in );
    System.out.print( "Enter start: " );
    start = keyboard.nextInt();
    for ( value = start; value > 0; value /= 2 )
    {
      System.out.println( ".." + value );
    }
    System.out.println( "BOOOOOOOOOOOOOOOOOOOOOOOOM!" );
  }
}
```

Listing 7.9 A program that counts down from an input number to 0 by dividing the number by 2

Here is one execution:

```
Enter start: 1000
..1000
..500
..250
..125
..62
..31
..15
..7
..3
..1
BOOOOOOOOOOOOOOOOOOOOOOOOM!
```

Here is another:

```
Enter start: 10000
..10000
..5000
..2500
..1250
..625
..312
..156
..78
..39
..19
..9
..4
..2
..1
BOOOOOOOOOOOOOOOOOOOOOOOOOM!
```

The next program receives the value for an `int` variable, `n`, from the user, and then computes the sum of integers between 1 and `n`. In Chap. 2, we have seen the program performing this calculation, using Gauss's approach for a fixed value of `n` (as `n * (n - 1 ) / 2`). The version here computes the sum by adding the numbers one after another:

```
import java.util.*;
public class LimitedGauss
{
  public static void main( String[] args )
  {
    int n, sum;
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter n: " );
    n = keyboard.nextInt();
    sum = 0;
    for ( int i = 1; i <= n; i ++ )
    {
      sum += i;
    }
    System.out.println( "The sum = " + sum );
  }
}
```

Listing 7.10 A program that calculates the sum of all integers between 1 and `n` by adding one number after another

The iteration variable `i` produces the sequence `1, ..., n` for a positive integer `n`. The program adds the value of `i` to a variable, `sum`, whose initial value is 0. Thus, the value of `sum` becomes 1, 3, 6, 10, and so on, ending with `n*(n-1)/2`.

Here is one execution example:

```
Enter n: 10
The sum = 55
```

Note that if the value of `n` is less than or equal to 0, the loop-body is never executed, and so the value `sum` retains its initial value, 0:

```
Enter n: -10
The sum = 0
```

In a similar vein, we can write a program that computes the product of all integers between 1 and `n`, where `n` is a value that the user enters. The mathematical name for the product of consecutive numbers starting from 1 is the *factorial* of `n`, written *n*!. Since the factorial is a function that increases very quickly as the value of `n` increases, we use `long` instead of `int`. We compute the value of the factorial in a `long` variable named `product`. Then, we initialize `product` with the value of 1. During the loop, we update the value of `product` by multiplying it with the value of the iteration variable `i`.

```
import java.util.*;
public class Factorial
{
  public static void main( String[] args )
  {
    int n;
    long product;
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter n: " );
    n = keyboard.nextInt();
    product = 1;
    for ( int i = 1; i <= n; i ++ )
    {
      product *= i;
    }
    System.out.println( "The product = " + product );
  }
}
```

Listing 7.11 A program that computes the factorial function using a for-loop

Here is one execution example:

```
Enter n: 34
The product = 4926277576697053184
```

Note that if the value of `n` is less than or equal to 0, the loop-body is never executed, and so the value `product` retains its initial value, 1, as shown next:

```
Enter n: -10
The product = 1
```

The next program receives values for two variables, `start` and `end`, from the user. Using these two values, the program computes the sum of all numbers of the form `start + 3 * i` that fall in the range from `start` to `end`. At the end, the program prints the sum.

```
1  import java.util.Scanner;
2  public class SumEveryThird
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard;
7      int sum, count, start, end;
8      keyboard = new Scanner( System.in );
9      System.out.print( "Enter start and end: " );
10     start = keyboard.nextInt();
11     end = keyboard.nextInt();
12     //-- initialize the sum
13     sum = 0;
14     //-- iterate the value of j from 1 to 100
15     for ( count = start; count <= end; count += 3 )
16     {
17       sum += count;
18     }
19     //-- output the result
20     System.out.println( "the sum = " + sum );
21   }
22 }
```

Listing 7.12 A program that computes the sum of every third number within a range

We use two successive calls to `nextInt` (Lines 8 and 9) to receive the input value of the user. The user may hit the return key after entering the first number, but at that point the program will still be waiting for the second number, meaning nothing will happen. After receiving the two numbers, the program executes an iteration with the initialization of `count = start` and the termination condition of `count <= end` (Line 15).

The execution of the code with three different pairs of input results is shown next.

```
1  Enter start and end: 100 10000
2  the sum = 16670050
```

```
1  Enter start and end: 599 599
2  the sum = 599
```

```
1  Enter start and end: 1000
2  100
3  the sum = 0
```

Note the following:

- In the second case, the loop-body executes exactly once, since `start` is equal to `end`.
- In the last case, the value of `end` is strictly smaller than the value of `start`, so the loop terminates before executing the loop-body.

7.2.2 Iteration with an Auxiliary Variable

The sequences that we produced through of iteration had no repetitions and were either monotonically increasing or monotonically decreasing. Is it possible to use a simple iteration to generate a sequence containing repetitions? Is it possible to use a simple iteration to generate a sequence that is not monotonic?

The answer to both questions is “yes”. Such sequences can be generated using an auxiliary variable. For examples, consider the sequence

[0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]

For all nonnegative integers `k`, `3k / 3`, `(3k + 1) / 3`, and `(3k + 2) / 3` are all equal to `k`. Therefore, the above sequence is identical to `a / 3` for `a = 0..14`. We can then generate the sequence in an auxiliary variable, `i`, as follows:

```
1 int i, a;
2 for ( a = 0; a <= 14; a ++ )
3 {
4   i = a % 3;
5 }
```

By attaching an output statement to the body of the loop:

```
1 int i, a;
2 for ( a = 0; a <= 14; a ++ )
3 {
4   i = a / 3;
5   System.out.print( " " + i );
6 }
7 System.out.println();
```

we can verify the output:

```
0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

In the above, if we use `%` instead of `/`, we generate the sequence:

```
0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

Furthermore, by changing the operation to `i = ( a % 3 ) - 1`, we have the following code:

```
1 int i, a;
2 for ( a = 0; a <= 14; a ++ )
3 {
4   i = ( a % 3 ) - 1;
5   System.out.print( " " + i );
6 }
7 System.out.println();
```

This code produces the sequence:

```
-1 0 1 -1 0 1 -1 0 1 -1 0 1 -1 0 1
```

7.3 Double For-Loops

The double for-loop is a special for-loop structure that comprises of two for-loops, with one loop appearing inside the other. Here is the general structure of the double for-loop:

```
for ( INITIALIZATION1; CONDITION1; UPDATE1 )
{
  SOME_CODE;
  for ( INITIALIZATION2; CONDITION2; UPDATE2 )
  {
    LOOP_BODY;
  }
  SOME_OTHER_CODE;
}
```

We call the first loop the **external loop** and the second loop the **internal loop**.

Suppose we have the following double for-loop:

```
int start1, start2, end1, end2, diff1, diff2, count1, count2;
...
for ( count1 = start1; count1 <= end1; count1 += diff1 )
{
  for ( count2 = start2; count2 <= end2; count2 += diff2 )
  {
    System.out.println( count1 + ":" + count2 );
  }
}
```

The code uses two iteration variables, `count1` and `count2`. Iteration by `count1` follows values `start1`, `start1 + diff1`, `start1 + 2 * diff1`, etc., until it reaches a value greater than `end1`. For each value of `count1`, a similar iteration takes places for `count2`.

```
import java.util.Scanner;
//--- print a rectangular box with "#"
public class DarkBox
{
  //-- main method
  public static void main( String[] args )
  {
    int height, width, vertical, horizontal;
    Scanner keyboard;
    keyboard = new Scanner( System.in );
    // set height and width
    System.out.print( "Enter height: " );
    height = keyboard.nextInt();
    System.out.print( "Enter width: " );
    width = keyboard.nextInt();
    // exterior loop
    for ( vertical = 1; vertical <= height; vertical ++ )
    {
      // interior loop
      for ( horizontal = 1; horizontal <= width; horizontal ++ )
      {
        System.out.print( "#" );
      }
      //-- newline is needed
      System.out.println();
    }
  }
}
```

Listing 7.13 A program that generates a rectangle drawn with hash marks

Double for-loops such as this one can be incorporated in the following example. The program receive two integers from the user and produces a rectangle made up of `'#'`s on the screen. The dimensions of the rectangle are equal to the numbers the user has entered.

The external loop is for counting the number of lines, and the internal loop is for counting the number of `'#'`s appearing in each line. Also, for both loops, if we change the start value to 1 and the termination condition to `<`, we still get the same result. This is because prior to the change, the vertical value iteration generates the sequence:

```
0, 1, 2, ..., height - 1
```

and the horizontal value iteration generates the sequence:

```
0, 1, 2, ..., width - 1
```

After the change, the vertical value iteration generates the sequence:

```
1, 2, ..., height
```

and the horizontal value iteration generates the sequence:

```
1, 2, ..., width
```

Here is an execution example of the code:

```
Enter height: 10
Enter width: 50
##################################################
##################################################
##################################################
##################################################
##################################################
##################################################
##################################################
##################################################
##################################################
##################################################
```

Here is another:

```
Enter height: 8
Enter width: 60
############################################################
############################################################
############################################################
############################################################
############################################################
############################################################
############################################################
############################################################
```

Here is one more execution example:

```
Enter height: 7
Enter width: 0
```

Since the width of the rectangle was 0, seven empty lines appeared.

The next code produces lines surrounding the rectangle, with `-` for the horizontal lines, `|` for the vertical lines, and `'+'` for the four corners.

The first part of the code consists of declarations and the user input. The values that the user enters are `height` and `width` (same as before).

```
import java.util.Scanner;
public class FramedBox
{
//--- print a rectangular box with "#" with a frame
  //-- main method
  public static void main( String[] args )
  {
    int height, width, horizontal, vertical;
    Scanner keyboard;
    keyboard = new Scanner( System.in );
    // set height and width
    System.out.print( "Enter height: " );
    height = keyboard.nextInt();
    System.out.print( "Enter width: " );
    width = keyboard.nextInt();
```

Listing 7.14 A program that generates a rectangle of hash marks with surrounding lines (part 1)

The second part consists of three components. The first component is for printing the top line, which is a `'+'`, a line of `'-'` having length `width`, and another `'+'` in the order of appearance. The second component is similar to the previous double for-loop, but has one `'|'` at the beginning and at the end. In addition, the program prints the second `'|'` with `System.out.println` instead of `System.out.print`. The third component is identical to the first one.

```
    // * * * * the top line * * * * //
    System.out.print( "+" );
    for ( horizontal = 1; horizontal <= width; horizontal ++ )
    {
      System.out.print( "-" );
    }
    System.out.println( "+" );
    // * * * * the middle lines * * * * //
    for ( vertical = 1; vertical <= height; vertical ++ )
    {
      System.out.print( "|" );
      for ( horizontal = 1; horizontal <= width; horizontal ++ )
      {
        System.out.print( "#" );
      }
      System.out.println( "|" );
    }
    // * * * * the bottom line * * * * //
    System.out.print( "+" );
    for ( horizontal = 1; horizontal <= width; horizontal ++ )
    {
      System.out.print( "-" );
    }
    System.out.println( "+" );
  }
}
```

Listing 7.15 A program that generates a rectangle of hash marks with surrounding lines (part 2)

Here is one example of executing the code.

```
java FramedBox
Enter height: 10
Enter width: 20
+--------------------+
|####################|
|####################|
|####################|
|####################|
|####################|
|####################|
|####################|
|####################|
|####################|
|####################|
+--------------------+
```

Specifying one of the two dimensions as 0 results in the following. This is when the height is 0:

```
Enter height: 0
Enter width: 10
+----------+
+----------+
```

and this is when the width is 0:

```
Enter height: 7
Enter width: 0
++
||
||
||
||
||
||
||
++
```

By using the value of the external iteration variable in the header of the internal iteration, it is possible to perform a complicated task. The next example shows this. The goal is to print an upside down triangle with the right angle at the upper-left corner, like this one:

```
**********
*********
********
*******
******
*****
****
***
**
*
```

The number of `'*'` printed in one line starts from some number and decreases one by one, until it becomes 1. The start number is equal to the number of `'*'` printed in the first line. Suppose an `int` variable named `down` contains the number of `'*'`s that must be in a give line. Using an iteration variable, `across`, the task of printing that specific line can be accomplished as:

```
      for ( across = 1; across <= down; across ++ )
      {
        System.out.print( "*" );
      }
      System.out.println();
```

By enclosing this for-loop in another for-loop that generates the sequence `10, 9, ..., 1` with the iteration variable `down`, the triangle can be generated:

```
    for ( down = 10; down >= 1; down -- )
    {
      for ( across = 1; across <= down; across ++ )
      {
        System.out.print( "*" );
      }
      System.out.println();
    }
```

We want this program to receive the length of the first line (or the number of lines) from the user. We will store the value in a variable named `height`. We modify the internal loop so that the last output of `"*"` is part of `println`. This reduces the number of times the inner loop-body is executed by 1.

These changes result in the following code:

```
import java.util.Scanner;
//-- print a triangle
public class TriangleFlipped
{
  //-- main method
  public static void main( String[] args )
  {
    int height, down, across;
    Scanner keyboard;
    keyboard = new Scanner( System.in );
    System.out.print( "Enter height: " );
    height = keyboard.nextInt();
    for ( down = height; down >= 1; down -- )
    {
      for ( across = 1; across <= down - 1; across ++ )
      {
        System.out.print( "*" );
      }
      System.out.println( "*" );
    }
  }
}
```

Listing 7.16 A program that prints an upside right-angled triangle of a given height

Here is an execution example of the code:

```
Enter height: 17
*****************
****************
***************
**************
*************
************
***********
**********
*********
********
*******
******
*****
****
***
**
*
```

7.4 Computing the Maximum and Minimum in a Series of Numbers

Here, we combine for-loops and conditional executions to write an application.

Our first application `MaxAndMin` receives a series of integer data from the user and computes the maximum and minimum of the series. Prior to receiving the series, the program asks the user to specify the series length: in other words, how many elements are in the series. The length, which is stored in a variable `nData`. The program then generates a sequence `1, ..., nData` using a for-loop and receives the elements in the series.

We use if-statements in three places.

First, we use it to check if `nData` is greater than 0. If `nData` is less than or equal 0, the program terminates itself by announcing a run-time error. The type of the run-time error is `IllegalArgumentException`. We generate the error with a special statement of `throw` in the following manner:

```
        throw new IllegalArgumentException(
            "\n    " + nData + " is not positive" );
```

The second line of the statement is interpreted as a `String` and printed as an error message before termination. Since it begins with `"\n "`, the error message will start in a fresh newline with four white space characters.

In general,

```
throw new ERROR_TYPE_NAME( MESSAGE );
```

is the syntax of terminating a program by generating a run-time error of type `ERROR_TYPE_NAME` with an error message of `MESSAGE`. The word `new` is a keyword indicating that this is a creation of an object data of type `ERROR_TYPE_NAME`, and `MESSAGE` is the actual parameter given to the constructor.

The second and third if-statements appear when the element that the user has entered is compared with the present maximum and minimum values for updates.

```
1  import java.util.*;
2  public class MaxAndMin
3  {
4    public static void main( String[] args )
5    {
6      int nData, max, min, input;
7      Scanner keyboard = new Scanner( System.in );
8      System.out.print( "Enter # of data: " );
9      nData = keyboard.nextInt();
10     if ( nData <= 0 )
11     {
12       throw new IllegalArgumentException(
13           "\n    " + nData + " is not positive" );
14
15     }
```

Listing 7.17 A program for computing the maximum and the minimum of an integer series (part 1). A part that is responsible for receiving the length and running a check

Next, the program receives the first element of the series and stores this in the variable `input`. Since this is the very first element, the program stores its value in both `maximum` and `minimum`. The declarations of all these variables appear in Line 6. If the value of `nData` is 1, then there will be no further input, and so the first number the user enters is both the maximum and the minimum.

```
16     System.out.print( "Enter Data No. 1: " );
17     input = keyboard.nextInt();
18     max = input;
19     min = input;
```

Listing 7.18 A program for computing the maximum and the minimum of an integer series (part 2). A part that is responsible for receiving the first number

Since the program has set the initial value to the variables for the maximum and the minimum, the for-loop iterates over `2, ..., nData` with the variable `round`. If we change the initialization to `round = 1`, the program will prompt the user to enter one extra element.

The prompt that the program uses in the for-loop is

```
"Enter Data No. " + round + ": "
```

This is consistent with the prompt

```
"Enter Data No. 1: "
```

that the program uses for the very first number. This way, the user sees no difference between the prompt for the first number and the prompt for the remaining numbers.

In the for-loop, the program compares the input that the user enters with the present maximum (Lines 25–28) and the minimum (Lines 29–32). If the input is greater than the maximum, the program stores the value of the input in the maximum. If the input is smaller than the minimum, the program stores the values of the input in the minimum. Note that we can substitute the `if` for the minimum with `else if`, because if the input is greater than the present maximum, then there is no way for the input to be smaller than the minimum.

Finally, the program uses `printf` to produce the result on the screen, using the placeholder `%d` for both the maximum and the minimum.

```
20      for ( int round = 2; round <= nData; round ++ )
21      {
22        System.out.print( "Enter Data No. " + round + ": " );
23        input = keyboard.nextInt();
24        if ( max < input )
25        {
26          max = input;
27        }
28        if ( min > input )
29        {
30          min = input;
31        }
32      }
33      System.out.printf( "max=%d, min=%d%n", max, min );
34    }
35  }
```

Listing 7.19 A program for computing the maximum and the minimum of an integer series (part 3). A part that is responsible for receiving the remaining numbers and printing the result

Here is an execution example where the run-time error `IllegalArgumentException` occurs:

```
1  Enter # of data: 0
2  Exception in thread "main" java.lang.IllegalArgumentException:
3      0 is not positive
4          at MaxAndMin.main(MaxAndMin.java:12)
```

The third line, `" 0 is not positive"` is the actual parameter of `throw new IllegalArgumentException`.

Here is an execution example of the code in which the calculation is successful:

```
1  Enter # of data: 8
2  Enter Data No. 1: -1546
3  Enter Data No. 2: 345
4  Enter Data No. 3: 98035
5  Enter Data No. 4: -876
6  Enter Data No. 5: 5121
7  Enter Data No. 6: 100001
8  Enter Data No. 7: -4
9  Enter Data No. 8: -200000
10 max=100001, min=-200000
```

7.5 A Betting Game

7.5.1 For-Loops with Skipped Execution

The next application is a simple betting game, where the user is asked to throw a die and bet which side is facing up.

There are two bet types: specify the exact number of dots or specify odd/even of the number. Here are some rules:

- The player starts with 50 chips and plays 10 times. However, if the player has lost all his/her chips, the game ends, regardless of which round the player is on.
- The number of chips that can be bet is between 1 and the number of chips the player possesses at that moment.
- After throwing a die, the following occurs:

  - If the player has bet on the exact number of dots and the side that has shown has the same number of dots, the player retains the bet chips and receives, as a reward, chips in the amount equal to five times the number of chips that have been bet.
  - If the player has bet on even/odd parity and the number of dots on the side that has shown has the same parity as the player's bet, the player retains the bet chips and receives, as a reward, chips in the amount equal to the number of chips that have been bet.
  - If neither is the case, the player loses the bet chips.

  We design the code using the following algorithm:

- A for-loop is used to repeat a single round of action ten times.
- The loop-body takes its action only if the number of chips in possession at the start of the body is strictly positive. The body consists of the following sequence events.
  - The player is advised of the number of chips currently in possession and asked the number of chips to be bet.
  - The player enters the bet amount. If it is less than or equal to 0, the amount is adjusted to 1 (the body being executed guarantees that the number of chips in possession is at least 1). If the bet amount is greater than the number of chips in possession, the bet amount is reduced to the number of chips is possession.
  - The player is asked to enter the type of bet: -1 for the odd parity, 0 for the even parity, and one of $1, \ldots 6$ for the exact number of dots.
  - The player enters the type. If the type is less than -1, then it is adjusted to -1, and if the type is greater than 6, then it is adjusted to 6.
  - Using `Math.random`, an `int` between 1 and 6 is generated to represent the number of dots on the face that is up.
  - The player is told if he/she has won the bet and is advised on the change has been made on the number of chips in possession.
- After completing the loop, the final number of chips in possession is reported.

Here is the program that implements this idea.

We use `int` variables `possession` to record the number of chips in possession, `betType` to record the bet type, `betAmount` to record the bet amount, and `number` to record the result after throwing the dice (Line 7).

```
1  import java.util.*;
2  public class BettingGame
3  {
4    public static void main( String[] args )
5    {
6      Scanner keyboard = new Scanner( System.in );
7      int betType, betAmount, number, possession = 50;
8
```

Listing 7.20 A betting game (part 1). The part that sets up the variables

Next is the for-loop and the beginning of its loop-body. The for-loop iterates the sequence `1, ..., 10` using an iteration variable, `i` (Line 9). For each round, an action is to be performed only if `possession` is positive (Line 11). The action is as follows:

- Inform the player of the round number and the current chip amount (Lines 13–15).
- Ask for the bet amount, receive it, and store it in `betAmount` (Lines 16 and 17).
- Make an adjustment to the amount if necessary as follows:

  - If its value is not positive, raise it to 1 (Lines 18–21);
  - if its value is greater than `possession`, reduce it to `possession`, since a player cannot bet more than what he/she has (Lines 22–25).
- After that, state the [possibly altered] bet amount (Line 26).

```
9     for ( int i = 1; i <= 10; i ++ )
10    {
11      if ( possession > 0 )
12      {
13        System.out.println( "================================" );
14        System.out.println( "This is round " + i );
15        System.out.println( "You have " + possession + " chips" );
16        System.out.print( "How much do you want to bet? " );
17        betAmount = keyboard.nextInt();
18        if ( betAmount < 1 )
19        {
20          betAmount = 1;
21        }
22        else if ( betAmount > possession )
23        {
24          betAmount = possession;
25        }
26        System.out.println( "Your bet amount is " + betAmount );
27
```

Listing 7.21 A betting game (part 2). The part that processes the bet amount

The next part is for determining the betting type. The program does the following:

```
28        System.out.print( "Enter your bet type: " );
29        System.out.println( "-1 for odd, 0 for even," );
30        System.out.print( "1..6 for an exact bet: " );
31        betType = keyboard.nextInt();
32        if ( betType < -1 )
33        {
34          betType = -1;
35        }
36        else if ( betType > 6 )
37        {
38          betType = 6;
39        }
40
```

Listing 7.22 Betting game (part 3). The part that determines the bet type

- Ask the player to enter the bet type (Lines 28–30).
- Receive the bet type and store the value in `betType` (Line 31).
- Make an adjustment if necessary:
  - If the type is less than -1, raise it to -1 (Lines 32–35);
  - if the type is greater than 6, reduce it to 6 (Lines 36–39).

Next comes the part where a die is thrown. This is the end of the big if-statement as well as the for-loop.

- Generate the value for `number` using the formula `1 + (int)( 6 * Math.random() )` (Line 41) and report this number (Line 42).
- Check win/loss and make adjustments as follows (Lines 43–58):
  - If either (`betType == -1` (odd) and `number % 2 == 1`) or (`betType == 0` (even) and `number % 2 == 0`), then the player has won an odd/even bet, so add `betAmount` to `possession` (Lines 43–48).
  - Otherwise, if `betType == number`, then the player has won an exact-face bet, so add `5 * betAmount` to `possession` (Lines 49–53).
  - Otherwise, subtract `betAmount` from `possession` (Lines 54–58).
- Line 59 is the end of the if-statement.
- Line 60 is the end of the for-loop.

```
41          number = 1 + (int)( 6 * Math.random() );
42          System.out.println( "The number is " + number );
43          if ( betType == -1 && number % 2 == 1 ||
44               betType == 0 && number % 2 == 0 )
45          {
46            System.out.println( "You've won!" );
47            possession += betAmount;
48          }
49          else if ( betType == number )
50          {
51            possession += 5 * betAmount;
52            System.out.println( "You've won big time!!!!!!" );
53          }
54          else
55          {
56            possession -= betAmount;
57            System.out.println( "You've lost!" );
58          }
59        }
60      }
```

Listing 7.23 A betting game (part 4). The part that throws a dice and reports the result

At the end of the ten rounds, the program concludes by reporting the final amount of `possession`.

```
61
62      System.out.println( "================================" );
63      System.out.println( "You ended with " + possession + " chips" );
64
```

Listing 7.24 A betting game (part 5). The part that produces the final reporting

The logical formula in Lines 43 and 44:

```
betType == -1 && number % 2 == 1 || betType == 0 && number % 2 == 0
```

is equivalent to:

```
betType + ( number % 2 ) == 0
```

as well as to:

```
Math.abs( betType ) == ( number % 2 )
```

7.5.2 The Statements `continue` and `break`

Two important statements that can be used in a for-loop body are `continue` and `break`. The statement `continue` instructs the program to skip the remainder of the loop-body and move on to the next round of the loop. The statement `break` instructs the program to terminate the execution of the loop immediately, ignoring the remainder of the present round and all the remaining rounds. If these statements appear inside an interior loop of multiple loops, their actions apply to the innermost loop that contains the statements.

Consider the following code block:

```
1  Scanner keyboard = new Scanner( System.in );
2  int sum = 0;
3  for ( int i = 1; i <= 30; i ++ )
4  {
5    if ( i % 7 == 0 )
6    {
7      continue;
8    }
9    int input = keyboard.nextInt();
10   if ( input == 0 )
11   {
12     break;
13   }
14   sum += i * input;
15 }
```

The program generates a sequence `1, ..., 30` with the iteration variable `i`, receives an input from the user if `i` is not a multiple of 7, and computes the sum of `i` multiplied by the input. However, if the user enters 0, the loop is terminated immediately.

Here is another example. In the following code, `a` is an `int` variable.

```
1  for ( int count = 1; count <= 100; count ++ )
2  {
3    a += 10;
4    System.out.println( count + ", " + a );
5    if ( a > 1000 )
6    {
7      break;
8    }
9  }
```

The program repeatedly increases `a` by 10 until the value of `a` exceeds 1000.

In the betting game program, we enclosed the entire action inside the for-loop in the if-statement whose condition was `possession > 0`, i.e.,

```
for ( int i = 1; i <= 10; i ++ )
{
  if ( possession > 0 )
  {
    THE_ACTION
  }
}
```

where `THE_ACTION` refers to the action to be performed. We can use `continue` to take `THE_ACTION` outside the if-block.

```
for ( int i = 1; i <= 10; i ++ )
{
  if ( possession > 0 )
  {
    continue;
  }
  THE_ACTION;
}
```

Once `possession` becomes 0, there will be no action to be performed in the loop-body. Therefore, we can use `break` to terminate the loop as soon as `possession` becomes 0.

```
for ( int i = 1; i <= 10; i ++ )
{
  THE_ACTION;
  if ( possession == 0 )
  {
    break;
  }
}
```

Because `possession` is decreased only at one location, we can move the if-statement containing the `break` statement inside after the statement for printing the message `"You've lost!"`.

```
for ( int i = 1; i <= 10; i ++ )
{
  ...
  else
  {
    possession -= betAmount;
    System.out.println( "You've lost!" );
    if ( possession == 0 )
    {
      break;
    }
  }
}
```

7.6 Computing the Fibonacci Sequence

The Fibonacci sequence $F_0, F_1, F_2, \ldots$ is an infinite integer sequence defined by: $F_0 = F_1 = 1$ and for all $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. The sequence is as follows:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Consider receiving an integer n from the user and printing the value of F_i for $i = 2, \ldots, n$. To determine the value of F_i, we only need the values of F_{i-1} and F_{i-2}. Therefore, we can accomplish the task by using a for-loop and memorizing just the two previous values in the sequence.

Using a for-loop that iterates the sequence `2, ..., n` with a variable named `i`, we compute the value of F_i. To record the values of F_i and its two predecessors, we use three `long` variables, instead of three `int` variables. This is because the values of the elements of the Fibonacci sequence increase very quickly. The three variables are `f`, `fp`, and `fpp`. They represent F_i, F_{i-1}, and F_{i-2} respectively. At the start of the loop body, it is ensured that the value of `fp` is equal to F_{i-1} and the value of `fpp` is equal to F_{i-2}. We obtain the value of F_i by `f = fp + fpp` and print its value. When the program returns to the start of the loop-body, the value of `i` has been increased by 1. Therefore, we must make sure that `fpp` and `fp` hold the values of F_{i-1} and F_{i-2} for the new value of `i`. This can be accomplished by first replacing the value of `fpp` with the value of `fp`, and then replacing the value of `fp` with the value of `f`. Also, before entering the loop, since the value of `i` starts with 2, we need to assign the value of F_1 to `fp` and F_0 to `fpp`.

The next code encapsulates the above ideas and prints the value of `f`, along with the values of `fp` and `fpp`.

```
1  import java.util.*;
2  public class FibonacciProgress
3  {
4    public static void main( String[] args )
5    {
6      long f, fp = 1, fpp = 1;
7      Scanner keyboard = new Scanner( System.in );
8      System.out.print( "Enter n: " );
9      int n = keyboard.nextInt();
10     for ( int i = 2; i <= n; i ++ )
11     {
12       f = fp + fpp;
13       System.out.println(
14           i + "\tf=" + f + "\tfp=" + fp + "\tfpp=" + fpp );
15       fpp = fp;
16       fp = f;
17     }
18   }
19 }
```

Listing 7.25 A program that computes the Fibonacci sequence

Note that the order of execution is (a) computing `f`; (b) printing the values of `i`, `f`, `fp`, and `fpp`; (c) updating `fpp`; (d) updating `fp`. Here is an execution example:

```
1  Enter n: 30
2  2: 2    fp=1, fpp=1
3  3: 3    fp=2, fpp=1
4  4: 5    fp=3, fpp=2
```

```
5: 8     fp=5, fpp=3
6: 13    fp=8, fpp=5
7: 21    fp=13, fpp=8
8: 34    fp=21, fpp=13
9: 55    fp=34, fpp=21
10: 89   fp=55, fpp=34
11: 144  fp=89, fpp=55
12: 233  fp=144, fpp=89
13: 377  fp=233, fpp=144
14: 610  fp=377, fpp=233
15: 987  fp=610, fpp=377
16: 1597         fp=987, fpp=610
17: 2584         fp=1597, fpp=987
18: 4181         fp=2584, fpp=1597
19: 6765         fp=4181, fpp=2584
20: 10946        fp=6765, fpp=4181
21: 17711        fp=10946, fpp=6765
22: 28657        fp=17711, fpp=10946
23: 46368        fp=28657, fpp=17711
24: 75025        fp=46368, fpp=28657
25: 121393       fp=75025, fpp=46368
26: 196418       fp=121393, fpp=75025
27: 317811       fp=196418, fpp=121393
28: 514229       fp=317811, fpp=196418
29: 832040       fp=514229, fpp=317811
30: 1346269      fp=832040, fpp=514229
```

After the first round, the value of `fp` is transferred to `fpp` and the value of `f` to `fp`.

Summary

- The format for a for-loop is:

  ```
  for ( INITIALIZATION; CONTINUATION CONDITION; UPDATE ) { ... }
  ```

  The initialization is the action to be performed before entering the loop. The update is the action to be performed after each execution of the loop-body.
- The declaration of the iteration variable may appear in the initialization part of a for-loop. The scope of an iteration variable whose declaration appears in the header of the loop is the entire loop.
- We formally call the process of generating a series of index values loop an "iteration". The variable we use in iteration is an iteration variable.
- A for-loop can be used for counting repetitions. The value of the iteration variable can be referred to in the loop-body to produce results that are dependent on the iteration variable.
- A double-loop is a loop inside a loop. Using a double-loop, one can manipulate a block of data that has two dimensions.
- The `continue` statement instructs the program to skip the present round of for-loop and move on the next round.
- The `break` statement instructs the program to terminate the execution of the for-loop immediately.

Exercises

1. **Evaluating a for-loop** How many lines of output will be generated by the code below?

```
for ( int i = 1; i < 1000; i *= 2 )
{
  System.out.println(i);
}
```

2. **Evaluating double loops** State the output generated by each of the following for-loops:

```
for ( int i = 0; i < 4; i ++ )
{
  for ( int j = i + 1; j <= 4; j ++ )
  {
    System.out.print( j );
  }
  System.out.println( "x" );
}
```

```
for ( int i = 0; i <= 4; i ++ )
{
  System.out.print( "=" );
  for ( int j = 0; j <= i; j ++ )
  {
    System.out.print( j );
  }
  System.out.println();
}
```

```
for ( int s = 5; s >= 1; s -- )
{
  for ( int t = 5; t > s; t -- )
  {
    System.out.print( t );
  }
  System.out.println( "@" );
}
```

```
for ( int s = 33; s >= 0; s -= 6 )
{
  for ( int t = 33 - s; t <= s; t += 5 )
  {
    System.out.print( t );
  }
  System.out.println( "#" );
}
```

3. **Double for-loop** Write a double for-loop that produces the following output:

```
123456789
3456789
56789
789
9
```

4. **Another double for-loop** Write a double for-loop that produces the following output:

```
987654321
8765432
76543
654
5
```

5. **Number iterations** For each of the number sequences below, write a for-loop with an iteration variable, `n`, that generates the sequence:
   (a) 1, 2, 3, 4, 5
   (b) 1, 10, 100, 1000, 10,000
   (c) 10, 8, 6, 4, 2, 0
   (d) 12, 9, 6, 3, 0, -3, -6, -9
6. **Generating a series of squares** Write a program named `Squares` that receives an integer from the user, stores the value in an `int` variable `n`, and uses a for-loop to produce the following output lines:

```
1
4
9
...
n^2
```

   We can expect the value the user enters to be positive. Noting that $(m+1)^2 - m^2 = 2m+1$, write an alternative version that keeps producing the squares by cumulatively adding odd integers to a variable that is initially 0.
7. **Printing a diamond of variable sizes** Write a program named `Diamond2` that receives an integer value from the user (that is guaranteed to be positive and even) and prints a filled diamond with the input value as the height and width. (For the shape below the input value is 10.) The program may run by calculating one half of the input number and using it in iteration.

```
    /\
   //\\
  ///\\\
 ////\\\\
/////\\\\\
\\\\\/////
 \\\\////
  \\\///
   \\//
    \/
```

8. **Number sequence with decreasing repetitions** Write a triple for-loop that produces the output below.

```
543210
554433221100
555444333222111000
55554444333322221111100000
5555544444333332222221111100000
555555444444333333222222211111110000000
55555554444444433333333222222221111111100000000
5555555554444444443333333332222222221111111110000000000
555555555544444444443333333333222222222211111111110000000000
```

9. **Number pyramid** Write a program that produces the output in the shape of a pyramid as shown next using one double for-loop:

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

10. **Output generated by for-loops 1** State the output the following code generates:

```
int gap = 1, n = 1;
for ( int count = 1; count <= 10; count ++ )
{
  System.out.println( n );
  n += gap;
  gap ++;
}
```

11. **Output generated by for-loops 2** State the output the following code generates:

```
int gap = 1, n = 1;
for ( int count = 1; count <= 10; count ++ )
{
  System.out.println( n );
  n += gap;
  gap += 2;
}
```

12. **Output generated by for-loops 3** State the output the following code generates:

```
int gap = 1, n = 1;
for ( int count = 1; count <= 10; count ++ )
{
  System.out.println( n );
  n += gap;
  gap *= 2;
}
```

13. **Output generated by for-loops 4** State the output the following code generates:

```
int gap = 10, n = 100;
for ( int count = 1; count <= 10; count ++ )
{
  System.out.println( n );
  n -= gap;
  gap --;
}
```

14. **Partial sums** Write a program, `PartialSums`, that receives an integer `top` from the user and returns the sum of all the sums `(m + ... + 2 * m)` such that `m` is between `1` and `top`. The return value must be 0 if `top` is less than equal to 0. By appropriately designing the components of the loops, this requirement can be met without having to check whether or not `top <= 0`.
15. **Double iteration, A** Write a program named `AllIJ1` that receives an integer `top` from the user and produces on the screen the output
`(i,j)`
for all values of `i` and `j` between 1 and `top`. Configure the loops so that in the output, the value of `i` does not decrease, and for each fixed `i`, the value of `j` is strictly decreasing.

Here is an execution example:

```
Enter one number: 4
(1,1)
(1,2)
(1,3)
(1,4)
(2,1)
(2,2)
(2,3)
(2,4)
(3,1)
(3,2)
(3,3)
(3,4)
(4,1)
(4,2)
(4,3)
(4,4)
```

16. **Double iteration, B** Write a program named `AllIJ2` that receives an integer `top` from the user and produces on the screen the output
`(i,j)`
for all values of `i` and `j` between 1 and `top` such that `i` is no greater than `j`. Configure the loops so that in the output, the value of `i` does not decrease, and for each fixed `i`, the value of `j` is strictly increasing.

Here is an execution example:

```
Enter one number: 4
(1,1)
(1,2)
(1,3)
(1,4)
(2,2)
(2,3)
(2,4)
(3,3)
(3,4)
(4,4)
```

17. **Double iteration, C** Write a program named `AllIJ3` that receives an integer `top` from the user and produces of the screen the output
`(i,j)`
for all values of `i` and `j` between 1 and `top` such that `i` is strictly less than `j`. Configure the loops so that in the output, the value of `i` does not decrease, and for each fixed `i`, the value of `j` is strictly increasing.

Here is an execution example:

```
Enter one number: 4
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
(3,1)
(3,2)
(3,3)
(4,1)
(4,2)
(4,3)
```

18. **Coordinated iteration** Suppose we want to print the coordinates of an N by N table, where N is greater than or equal to 2, and the rows and columns have indexes from 1 to N. The combination of a row number `x` and a column number `y` should be printed as `(x,y)`. For each row, the coordinates should be printed in one line. Write `CoordinatedIteration` that accomplishes this task. The program receives the value for N from the user. The output for `N = 3` is as follows:

```
Series 1
(1,1)(1,2)(1,3)
Series 2
(2,1)(2,2)(2,3)
Series 3
(3,1)(3,2)(3,3)
```

19. **Powers of a power** Write a program named `IntPowers` that receives three integers, a, p, and k, from the user and prints the values of a^p, a^{2p}, a^{3p}, $\cdots$, a^{kp}, where these values are `double` and appear one value per line. Use `Math.pow` to compute the powers.

Programming Projects

20. **The size of a toilet paper** Consider determining the length of toilet paper dispensed from a roll of toilet paper. To determine the length three parameters are needed: the radius of the core `coreRadius`, the maximum radius of the resulting roll of paper `maxRadius`, and the thickness of paper `thickness`. The way we will determine the length is as follows:
    - Each time the paper goes around the roll in one full circle, the radius increases by `thickness`. Therefore, for each `round >= 1`, if the paper goes around the roll `round` times, then the radius increases by `round * thickness`.
    - We want the resulting radius to be as large as possible, but not larger than `maxRadius`.
    - Based upon the above, we determine the exact number of times that the paper goes around the roll, represented by a variable `numberOfRounds`.
    - We assume that when the paper goes around the roll, which has radius `radius`, the length of the paper used for that particular round is 2π times `radius`.
    - The length of the roll generated is two times π times the sum of the radiuses of the roll from all rounds.

    Write a Java program named `ToiletPaper.java` that receives user `coreRadius`, `maxRadius`, and `thickness` from the user, and prints the length of the paper. Note that the integer part of any double `y` can be obtained by `(int)( Math.floor(y) )`.
21. **BMI for ranges of weights and heights** Write a program named `RangeBMI` that computes the BMI for a range of weights and heights. The user will specify the maximum and minimum

of the weights. He/she will also specify the maximum and minimum of heights. Furthermore, the user will specify the sizes of increments in weights and heights. The maximums, the minimums, and the increments are all integers. The program must use a double for-loop. In the external loop, the program generates an increasing sequence of weights to be considered. In the internal loop, the program generates an increasing sequence of heights to be considered. For each combination of weight and height, the program prints the weight, the height, and the BMI.

22. **Points on a circle** Write a program named `PointsOnCircle` that receives two positive quantities, a `double` data `myRadius` and an `int` data `myFraction`, from the user and prints the coordinates of the points on a circle with radius `myRadius`, whose rotation angles from the x-axis is between $-180°$ and $180°$ (with the degree incremented by `360 / myFraction`). The following is an example of running the program:

```
Enter the radius: 3
Enter the fraction: 8
(-3.6739403974420594E-16,-3.0)
(-2.121320343559643,-2.1213203435596424)
(-3.0,1.8369701987210297E-16)
(-2.1213203435596424,2.121320343559643)
(0.0,3.0)
(2.1213203435596424,2.121320343559643)
(3.0,1.8369701987210297E-16)
(2.121320343559643,-2.1213203435596424)
(3.6739403974420594E-16,-3.0)
```

23. **Computing the combinatorial numbers** Write a program named `Combinatorial` that computes combinatorial numbers after receiving two integers, n and k, from the user.

For integers $n \geq k \geq 0$, $C(n, k)$ is the number of possible ways to select k distinct elements from a group of n distinct elements. $C(n, k)$ is given as:

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

Here, $n!$ denotes the factorial of n. The factorial of n is $n(n-1)\cdots 1$ if $n \geq 1$ and 1 otherwise. Implement three ways to compute $C(n, k)$ on the values of n and k that the user enters.

(a) Write a method named `public static long factorial( int m )` that computes the factorial of m as a `long` data. Write a method named `combinatorial` that receives n and k as parameters and computes $C(n, k)$ using three calls to the method `factorial`.

(b) For all n and k, we know that $C(n, k) = C(n, n-k)$.

$$\frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots 1}{(k(k-1)\cdots 1)((n-k)(n-k-1)\cdots 1)}$$

Let q be the smaller of k and $n-k$. Then we have

$$\frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots 1}{(q(q-1)\cdots 1)((n-q)(n-q-1)\cdots 1)}$$

We can simplify the fraction to obtain

$$C(n, k) = \frac{n(n-1)\cdots(n-q+1)}{q(q-1)\cdots 1}$$

Write a method, `public static long product( int start, int end )`, that returns the product of all the integers between `start` and `end`, where `end >= start`. Write a method named `combinatorial2` that computes the two products using the method `product`.

(c) By reversing the order of appearance of the terms in the denominator $n(n-1)\cdots(n-q+1)$, we have

$$C(n,k) = \frac{n(n-1)\cdots(n-q+1)}{1\cdot 2\cdots(q-1)q}$$

Write an additional method named `combinatorial3` that computes the factorial. For $i = 1,\ldots,q$, `combinatorial3` executes the multiplication by the i-th term in the numerator and the division by the i-th term in the denominator.

The combinations of n and k for which the program computes $C(n,k)$ correctly increases, as follows:

```
1 Enter n and k: 20 10
2 With method 1, C(20,10)=184756
3 With method 2, C(20,10)=184756
4 With method 3, C(20,10)=184756
```

```
1 Enter n and k: 22 11
2 With method 1, C(22,11)=-784
3 With method 2, C(22,11)=705432
4 With method 3, C(22,11)=705432
```

```
1 Enter n and k: 40 20
2 With method 1, C(40,20)=0
3 With method 2, C(40,20)=-1
4 With method 3, C(40,20)=137846528820
```

24. **Treasure hunting** Write a program named `TreasureHunting`, that plays a game defined as follows: The player's goal is to find a treasure hidden at a location between 1 and 100. The player can make at most ten guesses. If the guess is correct, the program announces `"You have found the treasure!"` and halts. In every round, if the player makes an incorrect guess, based upon the distance between the true location and the guess, the program announces the following:

(a) If the distance is between 1 and 3, the program announces `"The treasure is very close."`.

(b) If the distance is between 4 and 6, the program announces `"The treasure is somewhat close."`.

(c) If the distance is greater than 6, the program announces `"The treasure is not close."`.

Starting in the second round, if the player makes an incorrect guess, the program informs the player whether the guess is closer than the previous one. The message printed is: `"You are closer."`, `"You are farther."`, or `"The same distance."`. If the user fails to find the treasure, the program should reveal the true location.