11 While-Loops and Do-While Loops

11.1 Using While-Loops

11.1.1 The Syntax of While-Loops

The while-loop is a loop that only requires a continuation condition. The structure of a while-loop is simple:

```
while ( CONDITION )
{
  STATEMENTS;
}
```

The meaning of this while-loop is "as long as `CONDITION` has the value of `true`, execute `STATEMENTS`". The diagram in Fig. 11.1 shows how a while-loop works. The for-loop and while-loop can simulate each other. First,

```
while ( CONDITION )
{
  STATEMENTS;
}
```

is equivalent to the for-loop without initialization and update, as shown next:

```
for ( ; CONDITION; )
{
  STATEMENTS;
}
```

Second,

```
for ( INITIALIZATION ; CONDITION; UPDATE )
{
  STATEMENTS;
}
```

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_11

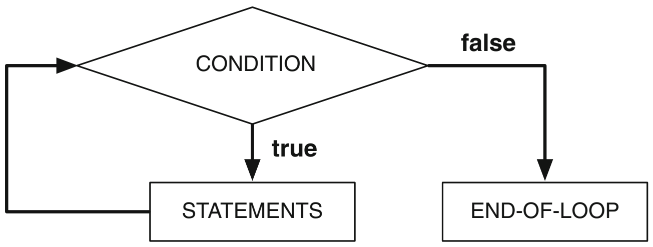

Fig. 11.1 A diagram that represents the while-loop

is equivalent to:

```
INITALIZATION;
while ( CONDITION )
{
  STATEMENTS;
  UPDATE;
}
```

We saw earlier that executing `next` on a `Scanner` object after obtaining its last token results in a run-time error of `NoSuchElementException`. In our example (List 3.1), we attempted to obtain five tokens from a `String` data with only four tokens, thereby intentionally produced the run-time error. It is possible to prevent an attempt to read beyond the last token by using the method `hasNext()` mentioned in Chap. 3. This method returns a `boolean` that represents whether or not at least one token is remaining.

The next program demonstrates the use of `hasNext` for reading all the tokens from an input `String` with an indefinite number of tokens. The method `hasNext` appears in the continuation condition (Line 7):

```
 1  import java.util.Scanner;
 2  public class BeyondLimitNew
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner tokens = new Scanner( "My age is 20" );
 7      while ( tokens.hasNext() )
 8      {
 9        String aToken = tokens.next();
10        System.out.println( aToken );
11      }
12    }
13  }
```

Listing 11.1 A program that reads all tokens from a `String` using a `Scanner` object

Here is an interactive version of the program, where the tokens are obtained from a `String` that the user enters:

The formal study of programming languages categorizes loops into two types, **definite loops** and **indefinite loops**, depending on whether the number of repetition is determined at the start of the loop or it is dynamically determined during the execution of the loop. The former type includes the for-loop and the latter includes the while-loop. Although for-loops and while-loops are interchangeable, there are situations where the use of one type appears more appropriate than the user of the other type.

```java
import java.util.Scanner;
public class TokenReadingWithInput
{
  public static void main( String[] args )
  {
    String aToken, tokens;
    Scanner keyboard = new Scanner( System.in );
    System.out.print( "Enter a String: " );
    tokens = keyboard.nextLine();
    while ( keyboard.hasNext() )
    {
      aToken = keyboard.next();
      System.out.println( aToken );
    }
  }
}
```

Listing 11.2 A program that reads all tokens from a `String` obtained from the user

11.1.2 Summing Input Numbers Until the Total Reaches a Goal

Our next example is a program that receives a series of integers from the user, and presents its running total after receiving each number. The program keeps receiving numbers from the user until the total reaches a preset goal. The program uses three `int` variables (`input`, `total`, and `goal`) to record the input from the user, the total of the input values, and the goal. The program initializes `total` with the value of 0 and uses the value of 1000 for `goal`. The process of receiving one integer and then adding its value to `value` can be code as follows:

```java
      input = keyboard.nextInt();
      total += input;
```

The program repeats this until the value of `total` exceeds `goal`. Using a while-loop, the repetition can be coded as follows:

```java
    while ( total <= goal )
    {
      input = keyboard.nextInt();
      total += input;
    }
```

Following is a source code of the program. The program uses a `Scanner` object `keyboard` instantiated with `System.in` (Line 6). The declarations of the `int` variables and their initializations appear in Line 7. In the while-loop, the program prints a prompt (Line 10), receives an input (Line 11), updates the total (Line 12), and reports the present values of `input` and `total` (Line 13) using `printf`. After exiting the loop, the program announces that the goal has been reached (Line 15).

Here is one execution example:

```
Enter input: 34
Input=34, Total=34
Enter input: 543
Input=543, Total=577
Enter input: 987
Input=987, Total=1564
The total has exceeded 1000.
```

```java
import java.util.*;
public class UpToLimit
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    int input, total = 0, goal = 1000;
    while ( total <= goal )
    {
      System.out.print( "Enter input: " );
      input = keyboard.nextInt();
      total += input;
      System.out.printf( "Input=%d, Total=%d%n", input, total );
    }
    System.out.printf( "The total has exceeded %d.%n", goal );
  }
}
```

Listing 11.3 A program that receives input numbers until the total exceeds a preset bound

The loop can be quickly finished by entering a large number:

```
Enter input: 355555
Input=355555, Total=355555
The total has exceeded 1000.
```

If the numbers entered, the loop lingers for a long time:

```
Enter input: 1
Input=1, Total=1
Enter input: 2
Input=2, Total=3
Enter input: 4
Input=4, Total=7
Enter input: 8
Input=8, Total=15
Enter input: 16
Input=16, Total=31
Enter input: 32
Input=32, Total=63
Enter input: 64
Input=64, Total=127
Enter input: 128
Input=128, Total=255
Enter input: 256
Input=256, Total=511
Enter input: 512
Input=512, Total=1023
The total has exceeded 1000.
```

11.1.3 Integer Overflow

The next example is a program that receives a positive initial value from the user, and then keeps updating the value with the multiplication by 2. The program uses an `int` variable named `number` to store the input number as well as the value after each update. Since the number of bits allocated for `int` is finite, the repeated doubling will eventually produce a value greater than the largest value an `int` can represent. We call such a phenomenon an **overflow**. If doubling is the cause of overflow, the value of the `int` variable immediately after the overflow is negative. Therefore, the program uses the continuation condition `while ( number >= 0 )` in the while-loop (Line 10). Lines 6–8 are responsible for receiving the initial value from the user. To ensure that the initial value is strictly positive, the program substitutes a nonpositive input with 1. This conditional substitution is accomplished using the assignment `number = Math.max( 1, number )` (Line 9). In the loop-body, the program reports the value of `number` (Line 12). An overflow will occur at Line 13, after reporting the value of `number` prior to doubling. Therefore, to report the negative value generated, the program has another statement to report the value of `number` (Line 15).

```java
 1  import java.util.*;
 2  public class IntegerOverflow
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner keyboard = new Scanner( System.in );
 7      System.out.print( "Enter number number: " );
 8      int number = keyboard.nextInt();
 9      number = Math.max( 1, number );
10      while ( number > 0 )
11      {
12        System.out.println( "The number is " + number );
13        number *= 2;
14      }
15      System.out.println( "The number is " + number );
16    }
17  }
```

Listing 11.4 A program that demonstrates an overflow with multiplication by 2

Here is one execution of the code:

```
Enter number: 35
The number is 35
The number is 70
The number is 140
The number is 280
The number is 560
The number is 1120
The number is 2240
The number is 4480
The number is 8960
The number is 17920
The number is 35840
The number is 71680
The number is 143360
The number is 286720
The number is 573440
The number is 1146880
The number is 2293760
The number is 4587520
The number is 9175040
The number is 18350080
The number is 36700160
The number is 73400320
The number is 146800640
The number is 293601280
The number is 587202560
The number is 1174405120
The number is -1946157056
```

The last value `-1946157056` is the result of multiplying `1174405120` by 2.

11.1.4 Vending Machines

Here is another example of the while-loop. In this example, we imitate the process of depositing coins into a vending machine for a purchase. Acceptable coins are the nickel (5 cents), the dime (10 cents), and the quarter (25 cents). While an actual vending machine takes one coin in one deposit, the program here takes multiple coins of the same kind in one deposit.

To receive multiple coins of a kind in one deposit, we design the program so that it asks the user to enter which coin she is depositing and how many. To specify the coin the user enters a `String` value. Depending on whether the `String` starts with `'N'`, `'D'`, or `'Q'`, the coin specification becomes the nickel, the dime, or the quarter. After receiving this input, the program sets the value of an `int` variable named `inc` to 5, 10, or 25, on the nickel, dime, and quarter. If the input is not one of the three, the program sets the amount of `inc` to 0. After this, if the value of `inc` is not 0, the program asks the user to enter how many coins of that kind she wants to deposit. The program increases the deposit by the product of this quantity and `inc`. The program repeats this two-step depositing process until the total deposit becomes greater than or equal to a target amount, `target`. `target` is an `int` constant with the value of 175 (that is, a dollar and 75 cents).

The program uses a while-loop to build the deposit:

```
while ( total < target )
```

To determine the value of `inc`, the program uses a switch statement:

```
    switch ( token.charAt( 0 ) )
    {
      case 'N': inc = 5; break;
      case 'D': inc = 10; break;
      case 'Q': inc = 25; break;
      default: inc = 0;
    }
```

Here, `token` is the input that the user enters when given a prompt for the coin type, and so `token.charAt( 0 )` is the very first character of the input. The switch statement examines this character and takes an appropriate action. The user may enter more characters after the first one, but those extraneous characters are ignored. After exiting the loop, the program reports the total deposit (Line 28) and announces that the user can make a purchase (Line 29).

Here is an execution example of the program:

```java
import java.util.*;
public class VendingDeposit
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    int total = 0, inc, count, target = 175;
    String token;
    while ( total < target )
    {
      System.out.printf( "Deposit is %d cents%n", total );
      System.out.println( "What coin do you deposit? " );
      System.out.print( "N for Nickel, D for Dime, Q for Quarter: " );
      token = keyboard.next();
      switch ( token.charAt( 0 ) )
      {
        case 'N': inc = 5; break;
        case 'D': inc = 10; break;
        case 'Q': inc = 25; break;
        default: inc = 0;
      }
      if ( inc > 0 ) {
        System.out.print( "How many? " );
        count = keyboard.nextInt();
        total += inc * count;
      }
    }
    System.out.printf( "Deposit is %d cents%n", total );
    System.out.println( "Now you can make a selection." );
  }
}
```

Listing 11.5 A program that mimics the process of depositing coins in a vending machine

```
Deposit is 0 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: N
How many? 5
Deposit is 25 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: D
How many? 3
Deposit is 55 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: N
How many? 2
Deposit is 65 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: Q
How many? 2
Deposit is 115 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: D
How many? 3
Deposit is 145 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: N
How many? 2
Deposit is 155 cents
What coin do you deposit?
N for Nickel, D for Dime, Q for Quarter: Q
How many? 2
Deposit is 205 cents
Now you can make a selection.
```

It is possible to determine the value of `inc` using a switch-statement with no `break` appearing in it, as follows:

```
    inc = 0;
    switch ( token.charAt( 0 )
    {
      case 'Q': inc += 15;
      case 'D': inc += 5;
      case 'N': inc += 5;
    }
```

The amount 15 for `'Q'` is the difference between a quarter and a dime, and the amount 5 for `'D'` is the difference between a dime and a nickel.

11.1.5 The Collatz Conjecture

Our next example is the program for testing the **Collatz Conjecture**, also known as the $3x + 1$ Problem, due to Lothar Collatz.[1] The conjecture states that every positive integer can be transformed to 1 by successively applying the following transformation:

(*) if the number is an even number, divide it by 2; otherwise, multiply it by 3 and then add 1 to it.

For example, 7 turns into 1 by following the procedure as follows:

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to$$
$$20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Portuguese scholar Tomás Oliveira e Silva has been conducting extensive research on this conjecture.[2] Through computer simulation, he has found that the conjecture is valid up to 5×2^{60}.

Consider receiving an input from the user and checking whether or not the conjecture holds for the input by repeatedly applying the transformation rule until the number becomes 1. We design the code using three methods, `update`, `method`, and `main`, as follows:

- The method `update` receives an integer as a formal parameter, applies the transformation rules *just once* to the number, and returns the number generated.
- The method `test` receives a positive integer as a formal parameter, and performs the test on the number. After each application of the rule, the method prints the number generated.
- The method `main` interacts with the user to receive input numbers, and then performs the test on them. The user can test the conjecture on an indefinite number of inputs. The program terminates when the user enters a non-positive number.

The source code of the program is shown next. After the program header, the method `update` appears. The formal parameter of the method is an `int` named `number`. The method returns an `int` value (Line 7). The actions to be performed by the method are as follows:

[1]Lothar Collatz (July 6, 1910 to September 26, 1990) was a German mathematician.

[2]The page of his work on this topic is: http://sweet.ua.pt/tos/3x_plus_1.html.

- The method checks whether or not the input number is a multiple of 2 with `number % 2 == 0` (Line 9).
- If the input number is a multiple of 2, then the method returns `number / 2` (Line 11).
- Otherwise, the method returns `number * 3 + 1` (Line 13).

Note that a `return` statement immediately terminates the execution of a method, so we do not need `else` in Line 13.

```
 1  // experimenting with Collatz Conjecture
 2  import java.util.*;
 3
 4  public class Collatz
 5  {
 6    // execute update and return the value
 7    public static int update( int number )
 8    {
 9      if ( number % 2 == 0 )
10      {
11        return number / 2;
12      }
13      return number * 3 + 1;
14    }
```

Listing 11.6 A program that tests the Collatz Conjecture (part 1). The method `update`

The method `test` has an `int` data named `input` as its formal parameter (Line 16). The method uses a variable named `number` and applies the transformation rule to the value stored in `number`. The initial value of `number` is equal to the value of `input`. The method uses a variable named `round` to record how many times the transformation rule has been applied to the input. The initial value of `round` is 0. The declarations and initializations of the two variables appear in Line 18. After reporting the value of `input` (Line 19), the method enters a while-loop with the continuation condition `number > 1` (Line 20). The loop terminates when the value of `number` becomes less than or equal to 1. In the loop-body, the method applies the transformation rule using the method `update`, replaces the value of `number` with the value returned (Line 22), adds 1 to `round` (Line 23), and reports the values of `round` and `number` (Line 24). To report the value of `round`, the method allocates 4 character spaces.

```
15    // test the conjecture
16    public static void test( int input )
17    {
18      int round = 0, number = input;
19      System.out.println( "input = " + input );
20      while ( number > 1 )
21      {
22        number = update( number );
23        round ++;
24        System.out.printf( "%4d: %d%n", round, number );
25      }
26    }
27
```

Listing 11.7 A program that tests the Collatz Conjecture (part 2). A part that performs the test on one number

We can expect the following different outcomes depending on the value of `input`:

- if `input <= 1`, the loop terminates without executing;
- if `input >= 2` and the conjecture holds for `input`, the loop eventually terminates;
- if `input >= 2` and the conjecture does not hold for `input`, the loop never terminates.

The method `main` uses a variable, `input`, to store the input from the user. The initial value of `input` is 1 (Line 33). The while loop in the method continues so long as `input >= 1` (Line 34). Therefore, the initial value of the variable makes the loop run. In the loop-body, the method obtains a new value for `input` (Lines 36 and 37). If the number entered is greater than or equal to 1 (Line 38), the method calls `test` (Line 40). After quitting the loop, the method prints a message (Line 43).

```
28     public static void main( String[] args )
29     {
30       System.out.println( "Testing the Collatz Conjecture" );
31       Scanner keyboard = new Scanner( System.in );
32 
33       int input = 1;
34       while ( input >= 1 )
35       {
36         System.out.print( "Enter a positive integer: " );
37         input = keyboard.nextInt();
38         if ( input >= 1 )
39         {
40           test( input );
41         }
42       }
43       System.out.println( "...quitting" );
44     }
45   }
```

Listing 11.8 A program that tests the Collatz Conjecture (part 3). The method `main`

Here are some execution examples of the program:

```
Testing the Collatz Conjecture
Enter a positive integer: 15
input = 15
   1: 46
   2: 23
   3: 70
   4: 35
   5: 106
   6: 53
   7: 160
   8: 80
   9: 40
  10: 20
  11: 10
  12: 5
  13: 16
  14: 8
  15: 4
  16: 2
  17: 1
```

```
Enter a positive integer: 17
input = 17
   1: 52
   2: 26
   3: 13
   4: 40
   5: 20
   6: 10
   7: 5
   8: 16
   9: 8
  10: 4
  11: 2
  12: 1
Enter a positive integer: 0
...quitting
```

11.1.6 Covnerting Decimal Numbers to Binary Numbers

The next program obtains the binary representation of a nonnegative decimal integer named `value`. The program uses the following logic:

- If `value` is equal to 0, then the representation is 0.
- Otherwise, initialize a `String` variable with an empty `String`, and then repeatedly divide `value` by 2, and insert the remainder of the division at the start of the `String` variable.

The next program incorporates this idea. The method `main` receives the input from the user (Line 14), stores it in an `int` variable named `number` (Line 15), and then calls the method `convert` (Line 16). `convert` is the method for executing the conversion. The method `main` stores the returned value in `result` (Line 16), and then prints the value of `result` (Line 17). This series of action is in a while-loop. The loop is repeated until the user enters a negative integer as the value to convert to binary.

The `convert` method receives an `int` value `number` as its formal parameter (Line 21). The method works as follows:

- If the value `number` is 0, the method returns `"0"` immediately (Lines 25).
- The method initializes a `String` variable named `binary` with the value of `""` (Line 27).
- The method repeats the following while `number > 0` (Line 28).
  - The method computes `number % 2` and stores it in a variable named `bit` (Line 30).
  - The method updates `binary` by inserting the value of `bit` (Line 31).
  - The method divides `number` by 2 (Line 32).
- After exiting the loop, the method returns `binary`.

The method modifies the value of the formal parameter `number`, but not the value of the actual parameter. If `number < 0`, the method returns an empty `String`.

```java
import java.util.*;

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

    int number = 0;
    String result;

    while ( number >= 0 )
    {
      System.out.print( "Enter an integer: " );
      number = keyboard.nextInt();
      result = convert( number );
      System.out.printf( "%d --> %s%n", number, result );
    }
  }

  public static String convert( int number )
  {
    if ( number == 0 )
    {
      return "0";
    }
    String binary = "";
    while ( number > 0 )
    {
      int bit = number % 2;
      binary = bit + binary;
      number /= 2;
    }
    return  binary;
  }
}
```

Listing 11.9 A program that obtains the binary representation of a nonnegative integer

Here is an execution example of the code:

```
Enter an integer: 0
0 --> 0
Enter an integer: 17
17 --> 10001
Enter an integer: 19
19 --> 10011
Enter an integer: 21
21 --> 10101
Enter an integer: 1027
1027 --> 10000000011
Enter an integer: 987654321
987654321 --> 111010110111100110100010110001
Enter an integer: -1
-1 -->
```

We can attach, to the conversion method, a line that shows the current value of `number` and the current value of `binary` during the course of the conversion while-loop.

```java
import java.util.*;

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

    long number = 0;
    String result;

    while ( number >= 0 )
    {
      System.out.print( "Enter an integer: " );
      number = keyboard.nextLong();
      result = convert( number );
      System.out.printf( "%d --> %s%n", number, result );
    }
  }

  public static String convert( long number )
  {
    if ( number == 0 )
    {
      return "0";
    }
    String binary = "";
    while ( number > 0 )
    {
      int bit = (int)( number % 2 );
      binary = bit + binary;
      number /= 2;
      System.out.printf( "%10d, %s%n", number, binary );
    }
    return  binary;
  }
}
```

Listing 11.10 A program that obtains the binary representation of a nonnegative integer while showing the progress on the calculation

Here are some execution examples with additional output:

```
Enter an integer: 9
    Digits, Binary
         4, 1
         2, 01
         1, 001
         0, 1001
9 --> 1001
```

```
Enter an integer: 1027
    Digits, Binary
       513, 1
       256, 11
       128, 011
        64, 0011
        32, 00011
        16, 000011
         8, 0000011
         4, 00000011
         2, 000000011
         1, 0000000011
         0, 10000000011
1027 --> 10000000011
Enter an integer: -1
-1 -->
```

11.1.7 Infinite Loops and Their Termination

An infinite loop is a loop that is designated to run forever. An infinite loop is constructed with `true` as the continuation condition. An infinite loop usually has a condition for termination inside the body. In Sect. 7.5.2, we saw the use of `break` to terminate a for-loop. `break` can be used to terminate a while-loop as well. Consider computing the number of times a pattern represented by a `String` variable named `pattern` appears in an input `String` variable named `input`. We can use an infinite loop for the task. We use a variable named `pos` to maintain the start position of the search in `input`. We also use a variable named `count` to record the number of occurrences of the pattern. The initial value is 0 for both `pos` and `count`. We use `input.indexOf( pattern, pos )` to check whether or not `pattern` appears in `input` at or after position `pos`. If the return value of `indexOf` is negative, we terminate the loop; otherwise, the return value is the position at which `pattern` appears, so we add one to `count` and update the value of `pos` with the returned value plus 1. The following code captures this idea:

```
  int count = 0, pos = 0;
  while ( true )
  {
    int res = input.indexOf( pattern, pos );
    if ( res < 0 )
    {
      break;
    }
    count ++;
    pos = res + 1;
  }
```

11.2 Using Do-While Loops

11.2.1 The Syntax of Do-While Loops

The do-while loop is a variant of while-loops, where the execution of the loop-body precedes the termination condition evaluation.

The structure of a do-while loop is:

```
do
{
  STATEMENTS
} while ( CONDITION );
```

The semicolon that appears is necessary.
We can rewrite a do-while loop using a while-loop. The following do-while loop

```
do
{
  STATEMENTS;
} while ( CONDITION );
```

is equivalent to

```
STATEMENTS;
while ( CONDITION )
{
  STATEMENTS;
}
```

Since a while-loop is an indefinite loop, we can write the program so that it will run forever using `true` as the termination condition.

```
while ( true )
{
  STATEMENTS;
  if ( CONDITION )
  {
    break;
  }
}
```

11.2.2 “Waiting for Godot”

Here is a simple program that uses a do-while loop.

Consider receiving a series of tokens from the user until the user enters “Godot”, when the execution terminates (where, of course, the “Godot” comes from a play by Samuel Beckett[3] titled *Waiting for Godot*). We store the user input to a `String` variable named `input`, and build a do-while loop using the condition `!input.equals( "Godot" )`. In other words, the program will run until the user enters `"Godot"`.

[3]Samuel Barclay Beckett (April 13, 1906 to December 22, 1989) was an Irish novelist and playwright. *Waiting for Godot* is an avant-garde play that features conversations between two men waiting for Godot, who never shows up.

If the code uses a while-loop, we need to assign some initial value other than the "Godot". Otherwise, the loop terminates immediately without asking the user to enter an input. If the code uses a do-while loop, the initialization is unnecessary.

Here is the version that uses a while-loop:

```java
import java.util.*;
public class Godot
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    String input = "";
    while ( !input.equals( "Godot" ) )
    {
      System.out.println( "This program is called \"Godot\"." );
      System.out.print( "Enter input: " );
      input = keyboard.nextLine();
    }
    System.out.println( "Terminating the program." );
  }
}
```

Here is the version that uses a do-while loop:

```java
import java.util.*;
public class DoWhileGodot
{
  public static void main( String[] args )
  {
    Scanner keyboard = new Scanner( System.in );
    String input;
    do
    {
      System.out.println( "This program is called \"Godot\"." );
      System.out.print( "Enter input: " );
      input = keyboard.nextLine();
    } while ( !input.equals( "Godot" ) );
    System.out.println( "Terminating the program." );
  }
}
```

11.2.3 Converting Decimal Numbers to Binary Numbers (Reprise)

Let us look back at the decimal-to-binary conversion program. In that program, a while-loop is used to repeat the interaction with the user indefinitely, and the loop was terminated when the user entered a negative value as an input. To prevent the loop from terminating without executing its body, the program assigned a nonnegative initial value to the variable. As in the case of the "Godot" program, if we use a do-while loop, such initialization is unnecessary.

```java
import java.util.*;

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

```

Listing 11.11 A program that obtains the binary representation of a nonnegative integer using a do-while loop (part 1)

```java
    int number;
    String result;

    do {
      System.out.print( "Enter an integer: " );
      number = keyboard.nextInt();
      result = convert( number );
      System.out.printf( "%d --> %s%n", number, result );
    } while ( number >= 0 );
  }

  public static String convert( int number )
  {
    if ( number == 0 )
    {
      return "0";
    }
    String binary = "";
    do
    {
      int bit = number % 2;
      binary = bit + binary;
      number /= 2;
      System.out.printf( "%10d, %s%n", number, binary );
    } while ( number > 0 );
    return  binary;
  }
}
```

Listing 11.12 A program that obtains the binary representation of a nonnegative integer using a do-while loop (part 2)

11.3 CTRL-D

Previously, to receive an indefinite number of input data from the user, we asked the user to enter a specific value to indicate the end of input. For example, in the decimal-to-binary conversion program, we asked the user to enter a negative integer to stop the program. Instead of using a special value, it is possible to use a special key to detect the end of input. The special key is called CTRL-D. The key can be entered by simultaneously pressing the "control" key and the 'D' key. When the method `hasNext` is called, if CTRL-D pressed before any other key, the method `hasNext` immediately returns `true`. Otherwise, `hasNext` waits until the return key is pressed, and then returns `false`.

Suppose `keyboard` is a `Scanner` object instantiated with `System.in`. We can write a while-loop of the following form:

```
while ( keyboard.hasNext() )
{
  READ_TOKEN_AND_REACT;
}
```

The next code demonstrates the use of this code. The program receives an indefinite number of integers from the user, and computes their total. The program accepts input until the user enters CTRL-D. The program uses three `int` variables, `input`, `total`, and `count`, for recording the input, the total, and the number of inputs. The last two variables have the initial value of 0. The program does not produce a prompt for individual inputs. It only announces at the beginning:

`"Enter inputs. End with a CTRL-D"`

The while-loop appears in Lines 9–14. At the end, the program prints the number of inputs entered and their total.

```
 1 import java.util.*;
 2 public class HasNextNoPrompt
 3 {
 4   public static void main( String[] args )
 5   {
 6     Scanner keyboard = new Scanner( System.in );
 7     int input, total = 0, count = 0;
 8     System.out.println( "Enter inputs.  End with a CTRL-D" );
 9     while ( keyboard.hasNext() )
10     {
11       input = keyboard.nextInt();
12       total += input;
13       count ++;
14     }
15     System.out.printf( "Count=%d, Total=%d%n", count, total );
16   }
17 }
```

Listing 11.13 A program for computing the total of integer input values

Here is one execution of the program. The inputs are: 10, 11, 12, 13, and 14. Each input line is followed by the return key. After entering 14 followed by the return key, the user presses CTRL-D. The CTRL-D does not echo.

```
Enter inputs.  End with a CTRL-D
10
11
12
13
14
Count=5, Total=60
```

Here is another execution. The user enters the same input numbers as before, but adds CTRL-D after some numbers. The CTRL-D entered after a number is ignored but echoes as `^D`.

```
Enter inputs.  End with a CTRL-D
10
11^D
12 ^D
13 ^D
14
Count=5, Total=60
```

The effect of CTRL-D as the end of input is permanent, meaning that once a `Scanner` object encounters CTRL-D as the result of `hasNext`, the result of `hasNext` will always be `false` and an attempt to read another token will result in a run-time error, `NoSuchElementException`. To resume reading from the keyboard, a new `Scanner` object must be instantiated with `System.in`.

The next program is a variant of the previous program. This time, the program produces the prompt `"Enter input: "` for each input number. The program accomplishes this by executing `System.out.print( "Enter input: " )` at two places (Lines 8 and 14). The one before entering the while-loop is necessary because the loop begins by waiting for an input.

```
 1  import java.util.*;
 2  public class HasNext
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner keyboard = new Scanner( System.in );
 7      int input, total = 0;
 8      System.out.print( "Enter input: " );
 9      while ( keyboard.hasNext() )
10      {
11        input = keyboard.nextInt();
12        total += input;
13        System.out.printf( "Input=%d, Total=%d%n", input, total );
14        System.out.print( "Enter input: " );
15      }
16      System.out.println( "\nEnd of the program." );
17    }
18  }
```

Here is an execution example of the code:

```
 1  Enter input: 340
 2  Input=340, Total=340
 3  Enter input: 35
 4  Input=35, Total=375
 5  Enter input: 98
 6  Input=98, Total=473
 7  Enter input: -180
 8  Input=-180, Total=293
 9  Enter input: ^D
10  End of the program.
```

In Line 9, the user enters CTRL-D, and the CTRL-D echoes as `^D`. Whenever CTRL-D is followed by the return key, it appear as `^D`. In the program, the ensuing return key appears in the `String` literal in Line 16.

To see the difference between having and not having the return key immediately after CTRL-D, let us compare the following two new programs. The two programs are variants of the previous program with slightly different Line 16: `HasNext01` prints `"...End of the program"` and `HasNext02` prints `"\n...End of the program"`.

```java
 1  import java.util.*;
 2  public class HasNext01
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner keyboard = new Scanner( System.in );
 7      int input, total = 0;
 8      System.out.print( "Enter input: " );
 9      while ( keyboard.hasNext() )
10      {
11        input = keyboard.nextInt();
12        total += input;
13        System.out.printf( "Input=%d, Total=%d%n", input, total );
14        System.out.print( "Enter input: " );
15      }
16      System.out.println( "...End of the program." );
17    }
18  }
```

```java
 1  import java.util.*;
 2  public class HasNext02
 3  {
 4    public static void main( String[] args )
 5    {
 6      Scanner keyboard = new Scanner( System.in );
 7      int input, total = 0;
 8      System.out.print( "Enter input: " );
 9      while ( keyboard.hasNext() )
10      {
11        input = keyboard.nextInt();
12        total += input;
13        System.out.printf( "Input=%d, Total=%d%n", input, total );
14        System.out.print( "Enter input: " );
15      }
16      System.out.println( "\n...End of the program." );
17    }
18  }
```

Here are execution examples of the two programs. In both executions, the user enters: 10, 20, and CTRL-D.

```
Enter input: 10
Input=10, Total=10
Enter input: 20
Input=20, Total=30
Enter input: ...End of the program.
Enter input: 10
Input=10, Total=10
Enter input: 20
Input=20, Total=30
Enter input: ^D
...End of the program.
```

11.4 Approximating the Square Root of a Real Number

Let's learn how to use a while-loop to approximate the square root of a number. Our goal is to write an application that receives a positive floating point number from the user, and then calculates an approximation of its square root. We will store the input in a variable named `original` and the approximated square root in a variable named `root`. To accomplish the goal, we use a simple strategy called **binary search**, that involves an indefinite loop. Later, we will see binary search in Sect. 13.5.2.

During the course of calculation, we manipulate two `double` variables, `high` and `low`. We ensure that the two variables have the following properties:

- The value of `high` is greater than or equal to the actual value of the square root.
- The value of `low` is smaller than or equal to the actual value of the square root.

In other words, the two properties are:

`high * high >= original` and `low * low <= original`.

We call a condition that is sustained throughout the execution of an indefinite loop a **loop invariant**. The conjunction of the above two properties is the loop invariant of the algorithm.

We initialize `high` with the value of `original + 0.5` and `low` with the value of 0. Since the square of `( original + 0.5 )` is equal to `original * original + original + 0.25` and `original` is positive, the first condition is met. Furthermore, since `original` is positive, the second condition is met.

In the do-while loop, we execute the following:

Step 1 We obtain a candidate for the square root. This is the half-way point between `high` and `low`, i.e., `( high + low ) / 2`. We store the candidate in `root`.
Step 2 We store the square of `root` in a variable named `square`.
Step 3 We store the value of `original - square` in another variable, `diff`.
Step 4 If `diff` is equal to 0, the value of `root` is the square root we are looking for, so we terminate the loop.
Step 5 If `diff` is positive, we update `low` with the value of `root`.
Step 6 If `diff` is negative, we update `high` with the value of `root`.
Step 7 We terminate the loop if `diff` becomes smaller than the value of a predetermined positive constant, `SMALL`.

The loop invariant is sustained when the value of `low` or `high` is updated. Furthermore, if the value of `low` or `high` is updated, then the value of `high–low` decreases to its half. We thus can anticipate that the value of `root` gets close to the actual value of the square root very quickly.

Based upon these observations, we hope that the loop will terminate eventually by either finding the actual value of the square root or finding that the value of `diff` has become smaller than the value of `SMALL`. However, this is not always the case. This is because the floating point number representation has a finite number of bits. To prevent such a situation from happening, we store the value of 10^{-12} in `SMALL`.

Next is the code of `SquareRoot` that implements the above idea. The declaration of the constant `SMALL` appears in Line 4. The input from the user is received in Line 9. The variables `high` and `low` are declared and initialized as planned in Line 10. We use an `int` variable `round` to record the number of rounds that have been executed (Line 12). The initial value of `round` is 0.

```
 1  import java.util.*;
 2  public class SquareRoot
 3  {
 4    public static double SMALL = 0.0000000000001;
 5    public static void main( String[] args )
 6    {
 7      Scanner keyboard = new Scanner( System.in );
 8      System.out.print( "> " );
 9      double original = keyboard.nextDouble();
10      double high = original + 0.5, low = 0;
11      double root, square, diff;
12      int round = 0;
```

Listing 11.14 A program for approximating the square root (part 1). The initialization of the variables

The continuation condition of the do-while loop is "the absolute value of `diff` is greater than or equal to `SMALL`". This is expressed as `diff >= SMALL || diff <= -SMALL` (Line 28). In the loop-body, the program adds 1 to the count (Line 15), stores the value of `( high + low ) / 2` in `root` (Line 16), and prints the values of `root` and `count`. The program then stores the square of `root` in `square` (Line 17) and stores the difference of the original from the square in `diff` (Line 18). Next, the program updates the values of `high` and `low` as follows: (Lines 20–27):

- If `diff > 0`, the program updates `low` with the value of `root`.
- If `diff < 0`, the program updates `high` with the value of `root`.

Finally, after exiting the loop, the program reports the values of the `root` and `square` (Line 29).

Here are execution examples of the program. First, we approximate the square root of 2.

```
> 2
Round=001,Value=1.250000000000000
Round=002,Value=1.875000000000000
Round=003,Value=1.562500000000000
Round=004,Value=1.406250000000000
Round=005,Value=1.484375000000000
Round=006,Value=1.445312500000000
Round=007,Value=1.425781250000000
Round=008,Value=1.416015625000000
Round=009,Value=1.411132812500000
Round=010,Value=1.413574218750000
Round=011,Value=1.414794921875000
Round=012,Value=1.414184570312500
Round=013,Value=1.414489746093750
Round=014,Value=1.414337158203125
```

```
    do
    {
      round ++;
      root = ( high + low ) / 2;
      System.out.printf( "Round=%03d,Value=%.15f%n", round, root );
      square = root * root;
      diff = original - square;
      if ( diff > 0 )
      {
        low = root;
      }
      else if ( diff < 0 )
      {
        high = root;
      }
    } while ( diff >= SMALL || diff <= -SMALL );
    System.out.printf( "Root=%.15f, Square=%.15f%n", root, square );
  }
}
```

Listing 11.15 A program for approximating the square root (part 2). The loop for approximation

```
Round=015,Value=1.414260864257813
Round=016,Value=1.414222717285156
Round=017,Value=1.414203643798828
Round=018,Value=1.414213180541992
Round=019,Value=1.414217948913574
Round=020,Value=1.414215564727783
Round=021,Value=1.414214372634888
Round=022,Value=1.414213776588440
Round=023,Value=1.414213478565216
Round=024,Value=1.414213627576828
Round=025,Value=1.414213553071022
Round=026,Value=1.414213590323925
Round=027,Value=1.414213571697474
Round=028,Value=1.414213562384248
Round=029,Value=1.414213557727635
Round=030,Value=1.414213560055941
Round=031,Value=1.414213561220095
Round=032,Value=1.414213561802171
Round=033,Value=1.414213562093210
Round=034,Value=1.414213562238729
Round=035,Value=1.414213562311488
Round=036,Value=1.414213562347868
Round=037,Value=1.414213562366058
Round=038,Value=1.414213562375153
Round=039,Value=1.414213562370605
Round=040,Value=1.414213562372879
Round=041,Value=1.414213562374016
Round=042,Value=1.414213562373448
Round=043,Value=1.414213562373163
Round=044,Value=1.414213562373021
Round=045,Value=1.414213562373092
Root=1.414213562373092, Square=1.999999999999992
```

Next, we approximate the square root of 3.

```
> 3
Round=001,Value=1.750000000000000
Round=002,Value=0.875000000000000
Round=003,Value=1.312500000000000
Round=004,Value=1.531250000000000
Round=005,Value=1.640625000000000
Round=006,Value=1.695312500000000
Round=007,Value=1.722656250000000
Round=008,Value=1.736328125000000
Round=009,Value=1.729492187500000
Round=010,Value=1.732910156250000
Round=011,Value=1.731201171875000
Round=012,Value=1.732055664062500
Round=013,Value=1.731628417968750
Round=014,Value=1.731842041015625
Round=015,Value=1.731948852539063
Round=016,Value=1.732002258300781
Round=017,Value=1.732028961181641
Round=018,Value=1.732042312622070
Round=019,Value=1.732048988342285
Round=020,Value=1.732052326202393
Round=021,Value=1.732050657272339
Round=022,Value=1.732051491737366
Round=023,Value=1.732051074504852
Round=024,Value=1.732050865888596
Round=025,Value=1.732050761580467
Round=026,Value=1.732050813734531
Round=027,Value=1.732050787657499
Round=028,Value=1.732050800696015
Round=029,Value=1.732050807215273
Round=030,Value=1.732050810474902
Round=031,Value=1.732050808845088
Round=032,Value=1.732050808030181
Round=033,Value=1.732050807622727
Round=034,Value=1.732050807419000
Round=035,Value=1.732050807520864
Round=036,Value=1.732050807571795
Round=037,Value=1.732050807546330
Round=038,Value=1.732050807559062
Round=039,Value=1.732050807565429
Round=040,Value=1.732050807568612
Round=041,Value=1.732050807570204
Round=042,Value=1.732050807569408
Round=043,Value=1.732050807569010
Round=044,Value=1.732050807568811
Round=045,Value=1.732050807568911
Round=046,Value=1.732050807568861
Root=1.732050807568861, Square=2.999999999999943
```

Summary

- A while-loop is a loop whose control requires the continuation condition only.
- A do-while loop is a variant of the while-loop, where the evaluation of the continuation condition occurs at the end of loop-body.
- An infinite loop takes the form of `while ( true ) { ... }`.
- It is possible to use an infinite loop with a mechanism for terminating the loop.
- CTRL-D is a key combination that indicates the end of input.
- The method `hasNext` of `Scanner` shows whether or not the CTRL-D is at the start of the input sequence.

Exercises

1. **A while-loop for printing numbers with leading 0s** Write a while-loop that produces the following output:

```
002:
004:
006:
008:
010:
```

2. **A do-while loop for printing two numbers per line** Write a do-while loop that produces the following output:

```
001.0,002.0
003.0,004.0
005.0,006.0
007.0,008.0
009.0,010.0
```

3. **A while-loop that produces a bit complex number output** Write a while-loop with a single `printf` statement inside the loop-body, and produces the following output:

```
+1024?
+0512?
+0256?
+0128?
+0064?
+0032?
+0016?
+0008?
+0004?
+0002?
+0001?
```

4. **Converting a for-loop to a do-while loop** Convert the following for-loop to an equivalent do-while loop:

```java
for ( int index = 1; index <= 15; index = index + 3 )
{
  System.out.println( index );
}
```

5. **Converting a for-loop to a while-loop** Convert the following for-loop to an equivalent while-loop:

```java
for ( int index = 10; index >= 0; index = index - 2 )
{
  System.out.println( index );
}
```

6. **Converting a while-loop to a do-while loop** Suppose that the following while-loop is part of a method that returns an `int`. Convert this loop to an equivalent do-while loop such that the `return` statement appears only after the do-while loop.

```java
Scanner keyboard = new Scanner( System.in );
int a, count = 0;
while ( true )
{
  count ++;
  System.out.print( "Enter an int: " );
  a = keyboard.nextInt();
  if ( a == 0 )
  {
    return count;
  }
}
```

7. **Substrings with while-loop** Suppose `word` is a `String` variable that is not `null`. Write a while-loop that produces all the nonempty suffixes of `word`, starting with the shortest one and ending with the longest one. Use an index variable that specifies the position where the suffix begins. For example, if `word` has the value `"sebastian-ibis"`, then the program must produce the following output:

```
s
is
bis
ibis
-ibis
n-ibis
an-ibis
ian-ibis
tian-ibis
stian-ibis
astian-ibis
bastian-ibis
ebastian-ibis
sebastian-ibis
```

8. **Random walk on a torus** A torus is a three-dimensional structure resembling a doughnut. A torus has a two-dimensional geometry. The points on the torus can be referred to using x- and y-coordinates. We consider here the case where both coordinates are integers between 0 and `boundary - 1` for some integer `boundary` greater than or equal to 2. For both dimensions, each coordinate value `v` has two neighbors. If `v` is between 1 and `boundary - 2`, the two neighbors are `v - 1` and `v + 1`. If `v` is equal to 0, the two neighbors are `boundary - 1` and `1`. If `v` is equal to `boundary - 1`, the two neighbors are `v - 1` and `0`. In this manner, each point on the grid has exactly four direct neighbors.

Consider the step-wise process where a point p that is located on such a grid randomly changes its location by moving to one of the four direct neighbors at each step. For each neighbor, the probability of moving to the neighbor is 25%. The initial location of p is the origin (0,0). Write a program, `RandomWalkTorus`, that simulates this process. The program must receive the value for `boundary` from the user and repeat the process until the point returns to the origin. After each step, the program must report the location of the point along with the number of steps it has executed.

The program may work as follows:

```
Enter the value for the boundary: 4
Round=1, Position=(3,1)
Round=2, Position=(2,2)
Round=3, Position=(3,1)
Round=4, Position=(0,0)
```

9. **A simple number generation** Using a do-while loop, write a program, `WaitForZero`, that repeats the following: randomly generate an integer between 0 and 9 and print the number generated, one per line. Use a do-while loop. Terminate the program when a 0 has been generated.
10. **A simple number generation plus** Using a while-loop, write a program, `WaitForThreeZeros`, that repeats the following: randomly generate an integer between 0 and 9 and print the number generated. The loop must be terminated when three 0's have been generated consecutively.
11. **Waiting for a pattern, no.1** Using a while-loop, write a program, `WaitForZeroOne`, that generates a random sequence of numbers between 0 and 9, and stops when a 1 is generated after a 0. The program must report each number generated in one character space. The numbers are printed in a single line, but after printing 60 characters, the program goes to the next line. Furthermore, after printing the required number of random digits, if the last line has few than 60 characters, the program goes to the next line before ending the program.

    The output of the program may look like this one:

```
% WaitForZeroOne
777330648687962933771263131311271180733257870261915397904279
313954419098344974921682097061190863460556325728765420418607
475305949271044338127842098973290463257596466804635930484292
1715082201
%
```

12. **Waiting for a pattern, no.2** Using a do-while loop, write a program `WaitForZeroOneTwo` that generates a random integer sequence of numbers between 0 and 9 and stops when the last three numbers generated become 0, 1, and 2 in this order. The program must report each number generated in one character space. The numbers are printed in a single line, but after printing 60 characters, the program goes to the next line. Furthermore, after printing the required number of random digits, if the last line has few than 60 characters, the program goes to the next line before ending the program.
13. **Guessing game** Write a program named `Guess` that plays a guessing game with the user. The program randomly selects an integer from $\{1, \ldots, 9\}$ using the code `(int)( Math.floor( 1 + Math.random() * 9 ) )`. The program then prompts the player to keep entering guesses. Each time the player enters a guess, the program must report the guess is correct or not. When the correct guess has been made, the program must terminate. Before terminating, the program must report how many guesses the player has made to arrive at the correct guess.

```
Enter your guess: 5
Your guess is not correct.
Enter your guess: 4
Your guess is not correct.
Enter your guess: 3
Your guess is not correct.
Enter your guess: 4
Your guess is not correct.
Enter your guess: 6
Your guess is correct.

You have found the answer with 5 guesses.
```

14. **Max and min simultaneously** Write a program named `MaxAndMin.java` that receives an indefinite number of real numbers from the user (as `double` values) and computes the maximum and the minimum of those that have been received. The program must terminate when the user enters 0, ignoring the 0 in the calculation of max and min. The program must use a `boolean` variable that is initialized with the value of `false`. Each time a number is received, the program must store `true` to this variable. During the execution of the loop, if the value of this variable is `false`, the program records the number that the user has entered as the maximum and minimum; if the value of this variable is `true` and the number the user has entered is not 0, the program compares the number with the maximum and the minimum to perform updates. After the loop, if the value of the `boolean` variable is `true`, the program reports the maximum and the minimum.
15. **Decimal to binary conversion** Write a program named `DecimalToBinaryWithBuilder` that receives a nonnegative `long` value from the user and produces its binary representation. Use a `StringBuilder` object in constructing the representation. The bit insertion must use the method `insert` of `StringBuilder`, and the return value must be the `String` that the builder represents. The program also must check whether the input from the user is nonnegative. If it is negative, the program must warn the user and halt immediately.
    Here is one execution example.

```
Enter a nonnegative integer: 3445844276438276431
The binary representation of 3445844276438276431 is
10111111010010000110010001110011010000011010101001100101001111
```

Programming Projects

16. **BMI with a while-loop** Write a program named `BMIWhile` that repeats the following: receive the weight and the height from the user, compute the BMI value determined by the two input values, and print the result on the screen (weight, height, and the BMI) with exactly two digits after the decimal point for each quantity. Use a while-loop in the code. Terminate the program when the user enters a nonpositive value for either weight or height.
17. **The area of triangle with a while-loop** Write a program named `TriangularAreaWhile` that repeats the following: receive three sides of a triangle, compute the area of the triangle specified by the sides, and print the result on the screen (the lengths, and the area). The program must use an infinite while-loop `while ( true )` and exit the loop if either one of the values entered is nonpositive or if one of the values entered is strictly greater than the sum of the other two (since such a triangle is impossible). The condition for exiting the loop must be the disjunction of six comparisons. To compute the area, use Heron's formula:

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a + b + c)/2$.

18. **The area of triangle with a while-loop, alternative** In the previous problem, we used the disjunction of six conditions as the condition for breaking the loop. Rewrite the program by replacing this condition with the disjunction of just two conditions with some pre-computation. Let `min` and `max` be the minimum and the maximum of the three sides. Then we have the following properties:
    - One of the sides is nonpositive if and only if `min` is nonpositive.
    - One side is longer than the sum of the other two if and only if two times `max` is greater than the sum of the three.
19. **Three-digit Mastermind** Mastermind is a game of two players. Mastermind is played as follows: Each player initially selects a four-digit number consisting of numerals 1 through 9 in which no digits appearing more than once. The players keep these numbers to themselves. After the initialization, the two players take turns in guessing the number the other player has. The player who guesses the number correctly first wins the game.

    When the opponent of a player makes a guess, the player having the secret number must answer whether or not the number matches the selected number; if not, the player must report the number of "hits" and the number of "misses", where a "hit" is a digit of the secret number appearing at exactly the same position in the guess and a "miss" is a digit of the secret number appearing at a different position in the guess. For example, for a secret number 9478, a guess 3417 has one hit (the 4) and one miss (the 7).

    We consider here a simplified version as follows:
    - The number of digits is not four but three.
    - The play is one-sided, meaning that only the first player selects a secret number and the second player tries to guess it correctly.
    - The first player is played by a computer program.

    Since no two digits can be equal in a secret number, the smallest possible number is 123 and the largest possible number is 987.

    Write a program named `MastermindSimple` that plays the role of the first year (the player that selects a secret number) as follows:
    (a) The program has a method

    ```
    public static boolean isLegit( int n )
    ```

    that checks whether or not the number `n` is between 123 and 987, none of the three digits of `n` are 0, and the three digits of `n` are pairwise distinct.
    (b) The program has a method

    ```
    public static int generate()
    ```

    that generates a random integer between 123 and 987 that passes the test. The method uses an infinite while-loop. In the loop, the method generates a random integer between 123 and 987, and returns the number if it passes the `isLegit` test.
    (c) The program has a method

    ```
    public static int countHit( int number1, int number2 )
    ```

    that returns, assuming that `number1` and `number2` have already passed the `isLegit` test, the number of hits when `number1` is a guess and `number2` is a secret.
    (d) The program has a method

    ```
    public static int countMiss( int number1, int number2 )
    ```

    that returns, assuming that `number1` and `number2` have already passed the `isLegit` test, the number of hits when `number1` is a guess and `number2` is a secret.

(e) The program uses either a while-loop or a do-while loop. In the loop, the program receives an integer guess from the user, stores the guess in an `int` variable named `guess`, and does the following:
   i It checks if the guess is legitimate using the `isLegit` method. If it is not legitimate, the program informs that the guess is not legitimate.
   ii Otherwise, if the guess is equal to the secret, the program congratulates the user and terminates the loop.
   iii Otherwise, the program counts the hits and the misses and reports the counts.

The program may work as follows:

```
Enter your guess: 123
No.Hits=0, No.Misses=0
Enter your guess: 456
No.Hits=2, No.Misses=0
Enter your guess: 457
No.Hits=1, No.Misses=1
Enter your guess: 467
No.Hits=0, No.Misses=2
Enter your guess: 478
No.Hits=0, No.Misses=1
Enter your guess: 567
No.Hits=0, No.Misses=3
Enter your guess: 756
Congratulations! You've guessed it right!
```

20. **Count the number of occurrences of "the"** Write a program, `CountThe`, that receives a series of `String` data from the user by way of `nextLine` of the class `Scanner`, concatenates the input lines into a single `String` data, and then in the concatenated data, counts the occurrences of the three-letter pattern `"the"`. To allow the user to enter any number of lines, use a while-loop that terminates when the user enters CTRL-D. To identify all the occurrences of `"the"`, use a variable that represents the start position of the search. The initial value of this variable is 0. When the search finds an occurrence at a position, the program updates the value of this variable with the position value plus 3 (the length of the literal `"the"`). The loop continues until either no more occurrence is found or the value of the variable representing the start position of search becomes greater than or equal to the length of the input.

    The program may run as follows:

```
Enter text (CTRL-D to stop) > Here is the program you need to write.
Enter text (CTRL-D to stop) > Your program solves the problem of finding
Enter text (CTRL-D to stop) > all the occurrences of the String literal
    "the"
Enter text (CTRL-D to stop) >  in the input character sequence that the
    user
Enter text (CTRL-D to stop) >  enters.
Enter text (CTRL-D to stop) > ^D
The number of occurrences is 7.
```

21. **Count the number of occurrences of either "the" or "an"** Extend the solution to the previous problem and write a program `CountTheAn` that computes the number of occurrences of either "the" or "an".