# Unit 6

# Loop statements

**Summary**

- Repetition of statements
- The `while` statement
- Input loop
- Loop schemes
- The `for` statement
- The `do` statement
- Nested loops
- Flow control statements

## 6.1 Statements in Java

Till now we have seen different types of statements (without counting declarations):

- simple statements:
  - method invocation
  - simple statements, i.e., assignment or use of an increment/decrement operator followed by `";"`
  - flow control statements (`break` and `return`)
- composite (or structured) statements
  - block of statements
  - conditional statements (`if-else`, `if`, `switch`)

## 6.2 Repetition of statements

Often, to solve a problem, it is necessary to repeat several times certain statement, possibly changing the value of some variable at each repetition.

Programming languages provide two ways to obtain the **repetition of statements**:

- the use of **loop statements** (or iterative statements), which are a form of composite statement;
- the use of *recursive methods*

We will see now the use of loops, and we will see later the use of recursion.

## 6.3 Definite and indefinite loops

Conceptually, we distinguish two types of loops, which differ in the way in which the number of *iterations* (i.e., repetitions of the body of the loop) is determined:

- In **definite loops**, the number of iterations is known before we start the execution of the body of the loop.

  *Example:* repeat for 10 times printing out a *.
- In **indefinite loops**, the number of iterations is not known before we start to execute the body of the loop, but depends on when a certain condition becomes true (and this depends on what happens in the body of the loop)

  *Example:* while the user does not decide it is time to stop, print out a * and ask the user whether he wants to stop.

In Java, like in other programming languages, both types of loop can be realized through a `while` statement.

## 6.4 The `while` loop

The `while` statement allows for the la repetition of a statement.

| `while` **statement** |
| --- |

*Syntax:*

```
while (condition)
  statement
```

- *condition* is an expression of type `boolean`
- *statement* is a single statement (also called the *body* of the loop)

*Note:* since, by making use of a block, it is possible to group several statements into a single composite statement, it is in fact possible to have more than one statement in the body of the loop.

*Semantics:*

- First, the *condition* is evaluated.
- If it is true, the *statement* is executed and the value of the *condition* is evaluated again, continuing in this way until the *condition* becomes false.
- At this point the statement immediately following the `while` loop is executed.

Hence, the body of the loop is executed as long as the *condition* stays true. As soon as it becomes false we exit the loop and continue with the following statement.

*Example:* Print out 100 stars.

```
int i = 0;
while (i < 100) {
  System.out.print("*");
  i++;
}
```

## 6.5 Use of a `while` loop for input

A `while` loop can be used to read from input a sequence of data, and to stop reading from input as soon as a certain condition becomes true. This is typically an *indefinite loop*.

*Example:* Read from input a set of strings and print them out on video until the user decides to stop.

```
String s = JOptionPane.showInputDialog("Input a string");
while (s != null) {
  System.out.println(s);
  s = JOptionPane.showInputDialog("Input a string");
}
```

Recall that `JOptionPane.showInputDialog("...")` returns `null` when the user presses the cancel button.

*Example:* Print the squares of integers read from input, until the user decides to stop.

```
int i;
String s;
s = JOptionPane.showInputDialog("Input an integer");
while (s != null) {
  i = Integer.parseInt(s);
  System.out.println(i*i);
  s = JOptionPane.showInputDialog("Input an integer");
}
```

**General structure of an input loop**

```
read the first element;
while (element is valid) {
  process the element;
  read the following element;
```

```
    }
```

## 6.6 Example of `while` loop: product through repeated sums

```
    int multiplicand, multiplicator, product;

    multiplicand = ...;
    multiplicator = ...;

    product = 0;
    while (multiplicator > 0) {
      product = product + multiplicand;
      multiplicator--;
    }
    System.out.println("product = " + product);
```

## 6.7 Example of `while` loop: division trough repeated subtraction

```
    int dividend, divisor, result, rest;

    dividend = ...;
    divisor = ...;

    result = 0;
    rest = dividend;
    while (rest >= divisor) {
      rest = rest - divisor;
      result++;
    }
    System.out.println("result = " + result);
    System.out.println("rest =   " + rest);
```

## 6.8 Example of `while` loop: power through repeated multiplication

```
    int base, exponent, power;

    base = ...;
    exponent = ...;

    power = 1;
    while (exponent > 0) {
      power = power * base;
      exponent--;
    }
    System.out.println("power = " + power);
```

## 6.9 Example of `while` loop: counting the occurrences of a character in a string

```
    public static int countChar (String s, char c) {
      int numchars = 0;
      int pos = 0;
      while (pos < s.length()) {
        if (s.charAt(pos) == c)
          numchars++;
        pos++;
      }
      return numchars;
    }
```

Note that in all four previous examples the loops are in fact *definite loops*, since the number of iterations depends only on the values of variables that have been fixed before starting the execution of the loop.

## 6.10   Characteristic elements in the design of a loop

```
initialization
while (condition) {
  operation
  next step
}
```

- *initialization*: sets the variables used in the loop before starting the execution of the loop (before the loop statement)
  *E.g.,* `product = 0;`
- *condition*: expression evaluated at the beginning of each iteration, whose truth value determines whether the body of the loop is executed or whether the loop finishes
  *E.g.,* `(multiplicator > 0)`
- *operation of the loop*: computation of the partial result at each loop iteration (in the body of the loop)
  *E.g.,* `product = product + multiplicand;`
- *next step*: increment/decrement operation for the variable that controls the repetitions of the loop (in the body of the loop)
  *E.g.,* `multiplicator--;`

Once we have designed the loop, we have to check its proper **termination**. In particular, we have to check that the execution of the loop statements can modify the value of the condition in such a way that it becomes false.

*Example:* The statement `multiplicator--;` can cause the condition `(multiplicator > 0)` to become false, provided `multiplicator` is a positive integer number.

## 6.11   Common errors in writing `while` loops

- *To forget to initialize a variable that is used in the condition of the loop.* Remember that the first time the condition is checked is before starting the execution of the body of the loop.
  *Example:*

  ```
  int i;
  String s;
  while (s != null) {
    i = Integer.parseInt(s);
    System.out.println(i*i);
    s = JOptionPane.showInputDialog("Input an integer");
  }
  ```

- *To forget to update the variables appearing in the condition of the loop.* The loop will never terminate.
  *Example:* The statement to read the next element to be processed is missing.

  ```
  int i;
  String s;
  s = JOptionPane.showInputDialog("Input an integer");
  while (s != null) {
    i = Integer.parseInt(s);
    System.out.println(i*i);
  }
  ```

- *Be off by 1 in the number of iterations.* Typically this is due to a mismatch between the condition of the loop and the initialization of the variables used in the condition.
  *Example:* Print out 10 stars.

  ```
  int i = 0;
  while (i <= 10) {            // should be (i < 10)
    System.out.print("*");
    i++;
  }
  ```

  To avoid such types of errors, it is often convenient to test the loop with simple (i.e., small) values for the variables. In the example above, if we check the loop by printing 1 star, instead of 10 (by substituting 10 with 1 in the condition of the loop), we immediately notice that the loop would print 2 stars, instead of 1.

## 6.12   Loop schemes

There are certain very common basic operations that require the use of loops:

- *counter*: count the number of values in a set;
- *accumulator*: accumulate values in a set according to a certain criterion;
- *characteristic values in a set*: determine a characteristic value among the values in a set (for example, the maximum, when the values in the set are ordered).

Each type of operation is based on a common scheme of the loop statement.

## 6.13   Loop scheme for a counter: number of strings in input

Count the number of times that a user inputs a string.

```
String s;        // current string in input
int counter;

counter = 0;
s = JOptionPane.showInputDialog("Input a string");
while (s != null) {
  counter++;
  s = JOptionPane.showInputDialog("Input a string");
}

System.out.println("Number of strings in input = " + counter);
```

## 6.14   Loop scheme for a counter: number of positive integers

Count the number of positive integers in input.

```
String s;        // current string in input
int counter;

counter = 0;
s = JOptionPane.showInputDialog("Input an integer");
while (s != null) {
  if (Integer.parseInt(s) > 0)
    counter++;
  s = JOptionPane.showInputDialog("Input an integer");
}

System.out.println("Number of positive integers in input = " + counter);
```

In this case, the increment of the counter depends on a condition.

## 6.15   Loop scheme for an accumulator: sum of integers

Sum the integers in input.

```
String s;        // current string in input
int n;           // current integer
int sum;         // variable used as accumulator

sum = 0;
s = JOptionPane.showInputDialog("Input an integer");
while (s != null) {
  n = Integer.parseInt(s);
  sum = sum + n;
  s = JOptionPane.showInputDialog("Input an integer");
}

System.out.println("sum = " + sum);
```

## 6.16   Loop scheme for an accumulator: product of integers

Multiply the integers in input.

```
String s;        // current input string
int n;           // current integer
int product;     // variable used as accumulator

product = 1;     // 1 is the neutral element for multiplication
s = JOptionPane.showInputDialog("Input an integer");
while (s != null) {
  n = Integer.parseInt(s);
  product = product * n;
  s = JOptionPane.showInputDialog("Input an integer");
}

System.out.println("product = " + product);
```

## 6.17   Loop scheme for an accumulator: concatenation of strings

Concatenate the strings in input that start with a ':'.

```
String s;         // current input string
String stot;      // variable used as accumulator

stot = "";           // "" is the neutral element for concatenation
s = JOptionPane.showInputDialog("Input a string");
while (s != null) {
  if (s.charAt(0) == ':')
    stot = stot + s;
  s = JOptionPane.showInputDialog("Input a string");
}

System.out.println("total string = " + stot);
```

## 6.18   Loop scheme for characteristic values in a set: maximum with a known interval

Let us consider the problem of finding the *maximum of a set of integers* in input. Assume that:

- we know the interval over which the integers range, in the sense that in the program we can denote the extremes of the interval (e.g., we know that all integers are greater than or equal to 0).

```
String s;        // current string in input
int n;           // current integer
int max;         // current maximum

max = -1;
s = JOptionPane.showInputDialog("Input an integer");
while (s != null) {
  n = Integer.parseInt(s);
  if (n > max) max = n;
  s = JOptionPane.showInputDialog("Input an integer");
}

if (max == -1)
  System.out.println("empty set of values");
else
  System.out.println("maximum = " + max);
```

*Note:* If the user does not input any integer, then the computed maximum value is -1. Since -1 is not in the interval of allowed values (we have assumed that all integers are greater than or equal to 0), it can never be

returned in the case where the user inputs at least one value. Therefore, in this case, we can use the comparison of the result with -1 as a means to detect whether the user has input at least a value or not.

## 6.19   Loop scheme for characteristic values in a set: maximum of a non-empty set

Let us consider the problem of finding the *maximum of a set of reals* in input. Assume that:

- the set of reals contains at least one value, but
- we do not know the interval over which the reals range, in the sense that in the program we cannot denote the extremes of the interval.

```
String s;        // current string in input
double r;        // current real
double max;      // current maximum

s = JOptionPane.showInputDialog("Input a real");
max = Double.parseDouble(s);
s = JOptionPane.showInputDialog("Input a real");
while (s != null) {
  r = Double.parseDouble(s);
  if (r > max)
    max = r;
  s = JOptionPane.showInputDialog("Input a real");
}

System.out.println("massimo = " + max);
```

*Note:* Since we know that the set of integers cannot be empty, we can read the first real and use it to initialize the maximum value with it before we start the loop.

## 6.20   Loop scheme for characteristic values in a set: maximum in the general case

Let us consider again the problem of finding the *maximum of a set of reals* in input. This time we make no assumption, i.e.,:

- the set of reals could be empty, and
- we do not know the interval over which the reals range.

In this case, a possible solution is the following.

```
String s;          // current string in input
double r;          // current real
double max = 0;    // current maximum
boolean found;     // indicates whether at least one value was input

found = false;
s = JOptionPane.showInputDialog("Input a real");
while (s != null) {
  r = Double.parseDouble(s);
  if (!found || (r > max)) {
    max = r;
    found = true;
  }
  s = JOptionPane.showInputDialog("Input a real");
}

if (found)
  System.out.println("maximum = " + max);
else
  System.out.println("empty set of values");
```

*Note:*

- If the set of reals contains at least one value, then the body of the `while` loop is executed at least once. At the first iteration, since the value of `found` is equal to false, the condition of the `if` statement is true

and the variable `max` is initialized to the value of the first real that has been input. In the subsequent iterations of the loop, since `found` is then equal to true (and hence `!found` is equal to false), the condition of the `if` statement will be true only if the current value of `r` is greater than `max`.

- If the set of input reals is empty, then the body of the loop will not be executed at all, and `found` keeps its value false.

- The initialization `max = 0;` would not be necessary for a correct execution of the program. Indeed, the expression `(f > max)` is evaluated only in the case where `found` is equal to true, and this happens only if the statement `max = f;`, which initializes `max`, has been executed. However, the Java compiler, which performs some checks on the initialization of variables, is not able to detect such a condition, and hence requires that `max` be initialized before evaluating the condition of the `if` statement.

There are other means for determining the maximum in the general case discussed here. For example, we could exploit the fact that the wrapper class `Double` provides the constant `MAX_VALUE` holding the maximum value a `double` can have, and initialize the maximum to `-MAX_VALUE`. We would anyway need a boolean variable to distinguish the case where the user has input no number at all from the case where the user has input just `-MAX_VALUE`.

```
String s;          // current string in input
double r;          // current real
double max;        // current maximum
boolean found;     // indicates whether at least one value was input

found = false;
max = -Double.MAX_VALUE;

s = JOptionPane.showInputDialog("Input a real");
while (s != null) {
  r = Double.parseDouble(s);
  found = true;
  if (r > max) max = r;
  s = JOptionPane.showInputDialog("Input a real");
}

if (found)
  System.out.println("maximum = " + max);
else
  System.out.println("empty set of values");
```

## 6.21   Other loop statements

In Java, there are three forms of loop statements:

- `while` loop
- `for` loop
- do loop

The `while` loop would be sufficient to express all loops that can be expressed with `for` or `do`. In certain situations, however, it is more convenient to develop an algorithm that makes use of the other types of loop.

## 6.22   Loop controlled by a counter

A common use of loops is the one in which the loop makes use of a variable (called *control variable*) that at each iteration is changed by a constant value, and whose value determines the end of the loop.

*Example:* Print the squares of the integers between 1 and 10.

```
int i = 1;
while (i <= 10) {
  System.out.println(i * i);
  i++;
}
```

The following are the common features of loops controlled by a counter:

- a *control variable* for the loop is used (also called *counter* or *indix* of the loop)
  *E.g.,* `i`
- *initialization* of the control variable
  *E.g.,* `int i = 1;`
- *increment* (or decrement) of the control variable at each iteration
  *E.g.,* `i++;`
- test if we have reached the *final value of the control variable*
  *E.g.,* `(i <= 10)`

The `for` statement loop allows to specify all these operations in a simple way:

*Example:* Print the squares of the integers between 1 and 10 using a `for` loop.

```
for (int i = 1; i <= 10; i++)
  System.out.println(i * i);
```

## 6.23 The `for` loop

<div align="center">

**for statement**
</div>

*Syntax:*

```
for (initialization; condition; update)
  statement
```

- *initialization* is an expression with side-effect that initializes a control variable (typically an assignment), which can also be a declaration with initialization
- *condition* is an expression of type `boolean`
- *update* is an expression with side-effect that typically consists in updating (i.e., incrementing or decrementing) the control variable
- *statement* is a single statement (also called *body* of the `for` loop)

*Semantics:* is equivalent to

```
{
  initialization;
  while (condition) {
    statement
    update;
  }
}
```

(There is an exception in the case of the `continue` statement, which needs to be translated in a more complicated way.)

*Example:* Print out 100 stars.

```
for (int i = 0; i < 100; i++)
  System.out.print("*");
```

## 6.24 Observations on the `for` loop

- If a control variable is declared in the *initialization* part, then its scope is limited to the `for` statement. This becomes clear if we look at the equivalent `while` loop, in which the entire code corresponding to the `for` loop is enclosed inside a block.
  *Example:*

```
for (int i = 0; i < 10; i++) {
  System.out.println(i * i);
}
// System.out.println("value of i = " + i);
                             // ERROR! i is not visible
```

- Each of the three parts of the `for` loop (i.e., *initialization*, *condition*, and *update*) could also be missing. In this case, the ";" have to be inserted anyway. If *condition* is missing, it is assumed to be equal to `true`.
- The syntax of the `for` loop allows all three parts to be arbitrary expressions, as long as *initialization*; and *update*; are statements (in particular, they must have a side-effect).

  However, when using the `for` loop the following is recommended:
    - Use the three parts of the `for` loop according to their intended meaning described above, and with reference to a *control variable* for the loop;
    - Do not modify the control variable in the body of the loop.
- In general, *initialization* and/or *update* can be a sequence of expressions with side-effect, separated by ",". This would allow us to initialize and/or update more than one control variable at a time. However, it is recommended not to do so.

  *Example:* Calculate and print the first 10 powers of 2.

  ```
  int i, powOf2;

  for (i = 0, powOf2 = 1; i < 10; i++, powOf2 *= 2)
    System.out.println("2 to the " + i + " = " + powOf2);
  ```

## 6.25   Examples of `for` loops

The `for` loop is used mainly to realize *definite loops*.

- `for (int i = 1; i <= 10; i++) ...`
  values assigned to i: 1, 2, 3, ..., 10
- `for (int i = 10; i >= 1; i--) ...`
  values assigned to i: 10, 9, 8, ..., 2, 1
- `for (int i = -4; i <= 4; i = i+2) ...`
  values assigned to i: -4, -2, 0, 2, 4
- `for (int i = 0; i >= -10; i = i-3) ...`
  values assigned to i: 0, -3, -6, -9

*Example:* Print the numeric code of the characters from 15 to 85.

```
for (int i = 15; i <= 85; i++) {
  char c = (char)i;
  System.out.println("i = " + i + " -> c = " + c);
}
```

## 6.26   Example of `for` loop: encoding a string

Write a public static method that takes as parameters a string and an integer $d$, and returns the encoding of the string with the integer. The encoding is obtained by substituting each character $c$ in the string with the character that has the code equal to the code of $c$ incremented by $d$.

*Example:* `"ciao"` with $d = 3$ becomes `"fldr"`

```
public static String encode(String s, int d) {

  String resStr;
  char c;
  int ci;

  resStr = "";
  for (int i = 0; i < s.length(); i++) {
    c = s.charAt(i);
    ci = (int)c;
    ci += d;
    c = (char)ci;
    resStr = resStr + c;
  }
   return resStr;
}
```

## 6.27   Example of `for` loop: "cracking" an encoded string

Suppose a public static method `decode` is available, which decodes a string encoded with the `encode` method we have seen before. Write a public static method that takes as parameters an already encoded string *str* and the minimum and maximum codes (respectively *min* and *max*) that could have been used for encoding *str*, and prints all strings obtained from *str* by decoding it with all values between *min* and *max*.

```
public static void crack(String str, int min, int max) {
  for (int i = min; i <= max; i++)
    System.out.println(decode(str, i));
```

## 6.28   The `do` loop

In a `while` loop, the condition of end of loop is checked at the beginning of each iteration. A `do` loop is similar to a `while` loop, with the only difference that *the condition of end of loop is checked at the end of each iteration.*

<div align="center">

**`do` statement**
</div>

*Syntax:*

```
do
  statement
while (condition);
```

- `condition` is an expression of type `boolean`
- *statement* is a single statement (also called the *body* of the loop)

*Semantics:* is equivalent to

```
statement;
while (condition)
  statement
```

Hence:

- First, the *statement* is executed.
- Then, the *condition* is evaluated, and if it is true, the *statement* is executed again, continuing in this way until the *condition* becomes false.
- At this point the statement immediately following the `do` loop is executed.

*Example:* Print out 100 stars.

```
int i = 0;
do {
  System.out.print("*");
  i++;
} while (i < 100);
```

## 6.29   Observations on the `do` loop

Since the condition of end of loop is evaluated only after the body of the loop has been executed, it follows that:

- *The body of the loop is executed at least once.* Hence, the `do` loop should be used in those cases where we need to repeatedly execute some statements, and would like that these statements are executed at least once.
- In general, it is not necessary to initialize the variable that appear in the loop condition before the loop. It is sufficient that these variables are initialized in the body of the loop itself (notice that this is different from `while` loops).

*Example:* Sum integers read from input until a 0 is read.

```
int i;
int sum = 0;
do {
  i = Integer.parseInt(JOptionPane.showInputDialog(
```

```
                            "Input an integer (0 to terminate)"));
    sum = sum + i;
  } while (i != 0);
  System.out.println("sum = " + sum);
```

Note that the syntax of the `do` statement requires that there is a 't' after `while (condition)`. To increase readability of the program, and in particular to avoid confusing the `while (condition);` part of a `do` loop with a `while` statement with empty body, it is in any case better to include the body of the `do` loop in a block, and indent the code as follows (as shown also in the example above):

```
  do {
    statement
  } while (condition);
```

## 6.30   Example of a `do` loop: input validation

Often it is necessary to validate data input by the user, and *repeat the request for the data in the case where the input of the user is not valid.* This can be done by using a `do` loop.

*Example:* Write a public static method that continues reading from input an integer until the integer is positive, and then returns the positive integer that has been input.

```
  public static int readPositiveInteger() {
    int i;
    do {
      i = Integer.parseInt(JOptionPane.showInputDialog(
                              "Input a positive integer"));
    } while (i <= 0);
    return i;
  }
```

Note that the previous method is not able to handle correctly all situations of incorrect input, for example, the situation where the user inputs an alphabetical character (or any sequence of characters that cannot be parsed by `parseInt`). We will see later on how Java allows us to handle such situations through the use of *exceptions*.

## 6.31   Equivalence between `while` loop and `do` loop

As is clear from the semantics, each `do` loop can be replaced with an *equivalent* `while` loop. However, to do so, we need to duplicate the body of the `do` loop.

*Example:*

```
  int i;

  do {
    i = Integer.parseInt(JOptionPane.showInputDialog(
                            "Input a positive integer"));
  } while (i <= 0);
```

equivale a

```
  int i;

  i = Integer.parseInt(JOptionPane.showInputDialog(
                          "Input a positive integer"));
  while (i <= 0) {
    i = Integer.parseInt(JOptionPane.showInputDialog(
                            "Input a positive integer"));
  }
```

## 6.32   Complete set of control statements

Two programs are said to be **equivalent** if, when they receive the same input,

- either both do not terminate, or
- both terminate and produce the same output.

---

©Diego Calvanese              Lecture Notes for Introduction to Programming              A.A. 2004/05

A set of control statements is said to be **complete** if, for each possible program that can be written in the programming language, there is an equivalent one that contains only the control statements in the set.

**Theorem by Böhm and Jacopini**

The following statements form a complete set: *sequencing*, `if` *statement*, and `while` *statement*.

## 6.33   Example: computing the greatest common divisor (GCD)

*Specification:*

We want to realize a static public method that takes as parameters two positive integers $x$ and $y$ and computes and returns their greatest common divisor $\gcd(x, y)$.

The greatest common divisor of two integers $x$ and $y$ is the greatest integer that divides both $x$ and $y$ without rest.

*Example:*   $\gcd(12, 8)$  =  4
$\gcd(12, 6)$  =  6
$\gcd(12, 7)$  =  1

## 6.34   GCD: by directly exploiting the definition

- We are looking for the maximum divisor of both $x$ and $y$.

- Observation: $1 \le \gcd(x, y) \le \min(x, y)$
  Hence, it is sufficient to try out the numbers between 1 and $\min(x, y)$.

- It is better to start from $\min(x, y)$ and go down toward 1. As soon as we find a common divisor of $x$ and $y$, we can immediately return it.

*First refinement of the algorithm:*

```
public static int greatestCommonDivisor(int x, int y) {
  int gcd;

  initialize gcd to the minimum of x and y
  while ((gcd > 1) && (we have not found a common divisor))
    if (gcd divides both x and y)
      we have found a common divisor
    else
      gcd--;

  return gcd;
}
```

*Observations:*

- The loop always terminates because at each iteration

    - either we find a divisor,

    - or we decrement `gcd` by 1 (at most we arrive at 1).

- To check whether we have found the gcd we make use of a boolean variable (used in the loop condition).

- To verify whether x (or y) divides `gcd` we use the `"%"` operator.

## 6.35   GCD: problems of the algorithm implementing directly the definition

How many times do we execute the loop in the previous algorithm?

- *best case:* 1 time, when $x$ divides $y$ or vice-versa
  E.g.,  gcd(500, 1000)

- *worst case:* $\min(x, y)$ times, when $\gcd(x, y) = 1$
  Ex . gcd(500, 1001)

Hence, the previous algorithm behaves bad when $x$ and $y$ are big and $\gcd(x, y)$ is small.

## 6.36   GCD: using the method by Euclid

The *method by Euclid* allows us to reach smaller numbers faster, by exploiting the following properties:

$$\gcd(x,y) = \begin{cases} x \quad (\text{or } y), & \text{if } x = y \\ \gcd(x - y, y), & \text{if } x > y \\ \gcd(x, y - x), & \text{if } x < y \end{cases}$$

This property can be proved easily, by showing that the common divisors of $x$ e $y$ are also divisors of $x - y$ (when $x > y$) or of $y - x$ (when $x < y$).

*E.g.,* $\gcd(12, 8) = \gcd(12 - 8, 8) = \gcd(4, 8 - 4) = 4$

To obtain an algorithm, we repeatedly apply the procedure until we arrive at the situation where $x = y$. For example:

| $x$ | $y$ | bigger $-$ smaller |
|-----|-----|--------------------|
| 210 | 63  | 147 |
| 147 | 63  | 84 |
| 84  | 63  | 21 |
| 21  | 63  | 42 |
| 21  | 42  | 21 |
| 21  | 21  | $\implies$ gcd(21,21) = gcd(21,42) = $\cdots$ = gcd(210,63) |

The algorithm can be implemented in Java as follows:

```
public static int greatestCommonDivisor(int x, int y) {
  while (x != y) {
    if (x > y)
      x = x - y;
    else                   // this means that y > x
      y = y - x;
  }
  return x;
}
```

## 6.37   GCD: admissible values for the arguments

What happens in the previous algorithm in the following cases:

- If $x = y = 0$?
  The result is 0.
- If $x = 0$ and $y > 0$ (or vice-versa)?
  The result should be $y$, but the algorithm enters an *infinite loop*.
- If $x < 0$ and $y$ is arbitrary (or vice-versa)?
  The algorithm enters an *infinite loop*.

Hence, if we want to take into account that the method could be called with arbitrary integer values for the parameters, it is necessary to insert a suitable test.

```
public static int greatestCommonDivisor(int x, int y) {
  if ((x > 0) && (y > 0)) {
    while (x != y)
      if (x > y)
        x = x - y;
      else                 // this means that y > x
        y = y - x;
    return x;
  } else
    System.out.println("wrong parameters");
}
```

## 6.38   GCD: using the method by Euclid with rests

What happens in the previous algorithm if $x$ is much bigger than $y$ (or vice-versa)?

*Example:*   gcd(1000, 2)                gcd(1001, 500)

| 1000 | 2 |
|---|---|
| 998 | 2 |
| 996 | 2 |
| $\vdots$ | $\vdots$ |
| 2 | 2 |

| 1001 | 500 |
|---|---|
| 501 | 500 |
| 1 | 500 |
| $\vdots$ | $\vdots$ |
| 1 | 1 |

To compress this long sequence of subtractions, it is sufficient to observe that we are actually calculating the rest of the integer division.

**Method by Euclid:** let $x = y \cdot k + r$ (with $0 \le r < y$)

$$\gcd(x, y) = \begin{cases} y, & \text{if } r = 0 \text{ (i.e., } x \text{ is a multiple of } y) \\ \gcd(r, y), & \text{if } r \ne 0 \end{cases}$$

The algorithm can be implemented in Java as follows:

```
public static int greatestCommonDivisor(int x, int y) {
  while ((x != 0) && (y != 0)) {
    if (x > y)
      x = x % y;
    else
      y = y % x;
  }
  return (x != 0)? x : y;
}
```

## 6.39   Example: length of the longest subsequence

Realize a public static method that takes as parameter a string $s$ (that is constituted only by the characters '0' and '1'), and returns the length of the longest subsequence of $s$ constituted only by consecutive '0's.

*Example:* If the string passed as parameter is `"001000111100"`, then the longest subsequence of only '0's is the underlined one, which has length 3.

```
public static int subsequence(String s) {
  char bit;          // current element in the sequence
  int cont = 0;      // current length of the sequence of zeros
  int maxlen = 0;    // temporary value of the maximum length

  for (int i = 0; i < s.length(); i++) {
    bit = s.charAt(i);

    if (bit == '0') {      // we have read a new '0'
      cont++;              // update the length of the current sequence
      if (cont > maxlen)   // if necessary, ...
                           // ... update the temporary maximum
        maxlen = cont;
    } else                 // we have read a 1
      cont = 0;            // reset the length of the current sequence
  }

  return maxlen;
}
```

## 6.40   Nested loops

The body of a loop can contain itself a loop, called a *nested loop*. It is possible to nest an arbitrary number of loops.

*Example:* Print out the multiplication table.

```
    public class MultiplicationTable {
      static final int NMAX = 10;

      public static void main (String[] args) {
        int row, column;

        for (row = 1; row <= NMAX; row++) {
          for (column = 1; column <= NMAX; column++)
            System.out.print(row * column + "  ");
          System.out.println();
        }
      }
    }
```

Note that we have used an integer constant `NMAX`, denoting the number of rows (and of columns) of the table. (The `static` keyword indicates that `NMAX` is not an instance variable, but a global variable for the class).

*Output produced by the program:*

```
1   2   3   4   5   6   7   8   9   10
2   4   6   8   10  12  14  16  18  20
3   6   9   12  15  18  21  24  27  30
4   8   12  16  20  24  28  32  36  40
5   10  15  20  25  30  35  40  45  50
6   12  18  24  30  36  42  48  54  60
7   14  21  28  35  42  49  56  63  70
8   16  24  32  40  48  56  64  72  80
9   18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

## 6.41   Example of nested loop: print out a pyramid of stars

In the previous example, where we printed out the multiplication table, the number of iterations of the internal loop was fixed. In general, the number of iterations of an internal loop may depend on the iteration of the external loop.

*Example:* Print out a pyramid of stars.

```
Pyramid of height 4       row   blanks    *
     *                     1      3       1
    ***                    2      2       3
   *****                   3      1       5
  *******                  4      0       7
```

To print the generic row $r$: print $(height - r)$ blanks and $(2 \cdot r - 1)$ stars.

```
    int height;
    height = Integer.parseInt(JOptionPane.showInputDialog("Input the height"));

    for (int row = 1; row <= height; row++) {
                        // 1 iteration for each row of the pyramid
      for (int i = 1; i <= height - row; i++)
        System.out.print(" ");     // prints the initial blanks

      for (int i = 1; i <= row * 2 - 1; i++)
        System.out.print("*");     // prints the sequence of stars

      System.out.println();        // prints a newline: the row is finished
    }
```

## 6.42   Example: power by means of a nested loop

```
    public static int power(int base, int exponent) {
      int result = 1;
      int multiplicand, multiplicator, product;
```

```
    while (exponent > 0) {
      exponent--;

      // result = result * base
      multiplicand = result;
      multiplicator = base;
      product = 0;
      while (multiplicator > 0) {
        multiplicator--;
        product = product + multiplicand;
      }
      result = product;
    }
    return result;
  }
```

## 6.43   Example: power by means of two methods

```
  public static int multiplication(int multiplicand,
                                   int multiplicator) {
    int product = 0;
    while (multiplicator > 0) {
      multiplicator--;
      product = product + multiplicand;
    }
    return product;
  }

  public static int power(int base, int exponent) {
    int result = 1;
    while (exponent > 0) {
      exponent--;
      result = multiplication(result, base);
    }
    return result;
  }
```

Note that in this case the internal loop is hidden inside the invocation of the `multiplication` method.

## 6.44   Flow control statements

Flow control statements determine the next statement to execute. In this sense, the statements `if-else`, `if`, `switch`, `while`, `for`, and `do` are flow control statements. However, these statements do not allow us to determine in an arbitrary way which is the next statement to be executed. Instead they *structure the program*, and the execution flow is determined by the structure of the program.

Java, as other programming languages, allows us to use (though with some limitations) also *jump statements*. Such statements are flow control statements that cause the interruption of the execution flow and the jumping to a statement different from the successive one in the sequence of program statements.

**Jump statements:**

- `break` (jump to the statement immediately following the current loop or `switch` statement)

- `continue` (jump to the condition of the loop)

*Note:*

- The use of jump statements should in general be avoided. They should be used only in specific situations. We will only mention them briefly.

- Also the `return` statement can be used to modify the execution flow, by causing a termination of the current method activation controlled by the programmer (see Unit 3).

## 6.45   Use of the `break` statement to exit a loop

We have already seen in Unit 5 that the `break` statement allows us to exit a `switch` statement. In general, *break allows us to exit prematurely from a `switch`, `while`, `for`, or `do` statement.*

*Example:* Loop to calculate the square root of 10 reals read from input. We want to interrupt the loop as soon as the user inputs a negative value.

```
double a;
for (int i = 0; i < 10; i++) {
  a = Double.parseDouble(
        JOptionPane.showInputDialog("Input a nonnegative real"));
  if (a >= 0)
    System.out.println(Math.sqrt(a));
  else {
    System.out.println("Error");
    break;
  }
}
```

*Note:* In the case of nested loops or of `switch` statements nested within a loop, the execution of a `break` causes the *exit from a single level of nesting only.*

## 6.46   Elimination of `break`

The execution of a `break` statement *modifies the execution flow*. Hence, when it is used in loops:

- we lose the structuring of the program;
- we may gain in efficiency with respect to implementing the same behavior without making use of `break`.

In general, it is always possible to **eliminate a `break` statement**. For example, the statement

```
while (condition) {
  statements-1
  if (break-condition) break;
  statements-2
}
```

is equivalent to

```
boolean finished = false;
while (condition && !finished) {
  statements-1
  if (break-condition)
    finished = true;
  else {
    statements-2
  }
}
```

The choice on whether to eliminate or not a `break` must be made by evaluating:

- on the one hand, the gain in efficiency for the program with `break` with respect to the program without it;
- on the other hand, the loss in readability due to the presence of `break`.

## 6.47   Example of elimination of a `break`

```
double a;
for (int i = 0; i < 10; i++) {
  a = Double.parseDouble(
        JOptionPane.showInputDialog("Input a nonnegative real"));
  if (a >= 0)
    System.out.println(Math.sqrt(a));
  else {
```

```
        System.out.println("Error");
        break;
      }
    }
```

is equivalent to

```
    double a;
    boolean error = false;

    for (int i = 0; (i < 10) && !error; i++) {
      a = Double.parseDouble(
            JOptionPane.showInputDialog("Input a nonnegative real"));
      if (a >= 0)
        System.out.println(Math.sqrt(a));
      else {
        System.out.println("Error");
        error = true;
      }
    }
```

### 6.48 The `continue` statement (optional)

The `continue` statement can be used only within loops, and its effect is to *directly jump to the next loop iteration*, skipping for the current iteration those statements that follow in the loop body.

*Example:* Print out the odd numbers between 0 and 100.

```
    for (int i = 0; i <= 100; i++) {
      if (i % 2 == 0)
        continue;
      System.out.println(i);
    }
```

Note that, when a `continue` statement is used in a `for` loop, the update-statement of the `for` loop is not skipped and is executed anyway.

*Note:* A possible use of `continue` is within a loop for reading from input, in which we want to execute some operations on the read data item only if a certain condition is satisfied. However, we have to make sure that at each iteration of the loop, the next data item is read in any case; otherwise the loop would not terminate.

*Example:* Wrong use of the `continue` statement.

```
    read the first data item;
    while (condition) {
      if (condition-on-the-current-data)
        continue;            // ERROR! the reading of the next data item is skipped
      process the data;
      read the next data item;
    }
```

### 6.49 Statements to exit from blocks and labels for statements (optional)

Normally, a `break` or a `continue` statement causes the exit from a single level of nesting of `switch` or loop statements in which it appears. However, such statements allow also for exiting from more than one level of nesting of a `switch` or loop.

To do so, the statements that define a block can have a **label**:

```
    label : loop-statement ;
```

A *label* must be a constant integer expression (analogous to those used in the ) `case`s of a `switch` statement).

The statement

```
    break label ;
```

interrupts the loop that has the label specified in the `break` statement. If there is no loop with the specified label that surrounds the `break` *label* statement, then a compile-time error is signaled.

*Note:* The use of labels and of `break` and `continue` statements that refer to labels is considered a bad programming habit, and has to be used only in very particular cases. In this course *we will not make use of it.*

## 6.50  Exercise: a class to encode texts

*Specification:*

Realize a Java class to represent encrypted texts. The encryption of the text is obtained by replacing each character with the character whose code is equal to the code of the character to encode augmented by an integer number representing the encryption key. The functionalities of encrypted texts are:

- creation of a new object that represents a text encrypted with a given encryption key;
- creation of a new object that represents a text that is not encrypted (yet);
- return of the encrypted text;
- return of the decrypted text, provided the correct encryption key is provided;
- verification of the correctness of an encryption key;
- modification of the encryption key; this is possible only if the correct key is provided.

*Solution scheme:*

To realize the Java class we make use of the methodology introduced in Unit 3.

The properties and the services of interest can be immediately identified from the specification. Hence, we can start writing:

```
public class EncryptedText {
  // private representation of the objects of the class
  // public methods that realize the requested functionalities
  // possibly auxiliary methods
}
```

In the following, we will choose the representation for the objects of the class, the public interface of the class, and the realization of the methods.

## 6.51  The class `EncryptedText`: representation of the objects

We have to decide how to represent the properties of encrypted texts. Note that an encrypted text needs two properties: the text itself and the encryption key. Let us represent the objects of the class `EncryptedText` by means of the following instance variables:

- the text, by means of an instance variable `text`, of type `String`;
- the key, by means of an instance variable `key`, of type `int`.

At this point we can write:

```
public class EncryptedText {
  // representation of the objects of the class
  private int key;
  private String text;

  // public methods that realize the requested functionalities
  // possibly auxiliary methods
}
```

## 6.52  The class `EncryptedText`: public interface

We can now choose the interface of the class, through which the clients can make use of the objects of the class `EncryptedText`.

Specifically, for each functionality we have to define a public method that realizes it and determine its header.

This leads us to the following skeleton for the class `EncryptedText`:

```
public class EncryptedText {
  // representation of the objects of the class
  private int key;
  private String text;

  // constructor
  public EncryptedText(String nonEncryptedText) {
        ...
  }
  public EncryptedText(String nonEncryptedText, int key) {
    ...
  }
  // other public methods
  public String getEncryptedText() {
        ...
  }
  public String getDecryptedText(int key) {
        ...
  }
  public boolean isKey(int candidateKey) {
        ...
  }
  public void setKey(int key, int newKey) {
        ...
  }

  // possibly auxiliary methods
}
```

## 6.53   The class `EncryptedText`: realization of the methods

We concentrate now on the single methods and realize their bodies. To do so, we make use of two auxiliary methods, `encode` and `decode`, analogous to the methods for encoding and decoding text according to a given key that we have already seen.

```
public class EncryptedText {
  // representation of the objects of the class
  private int key;
  private String text;

  // constructor
  public EncryptedText(String nonEncryptedText) {
    key = 0;
    text = nonEncryptedText;
  }
  public EncryptedText(String nonEncryptedText, int key) {
    this.key = key;
    text = encode(nonEncryptedText,key);
  }

  // altri metodi pubblici
  public String getEncryptedText() {
    return text;
  }
  public String getDecryptedText(int key) {
    if (key == this.key)
      return decode(text, key);
    else return null;
  }
  public boolean isKey(int candidateKey) {
    return candidateKey == key;
  }
```

```
        public void setKey(int key, int newKey) {
          if (key == this.key) {
            this.key = newKey;
            text = encode(decode(text,key),newKey);
          }
        }

        // auxiliary methods
        private static String encode(String text, int key) {
          String resText;
          char c;
          int ci;
          resText = "";
          for (int i = 0; i < text.length(); i++) {
            c = text.charAt(i);
            ci = (int)c;
            ci = ci + key;
                c = (char)ci;
            resText = resText + String.valueOf(c);
          }
          return resText;
        }
        private static String decode(String text, int key) {
          String resText;
          char c;
          int ci;
          resText = "";
          for (int i = 0; i < text.length(); i++) {
            c = text.charAt(i);
            ci = (int)c;
            ci = ci - key;
            c = (char)ci;
            resText = resText + String.valueOf(c);
          }
          return resText;
        }
      }
```

## 6.54   The class `EncryptedText`: example of a client

We realize the class `ClientEncryptedText`, which contains a methdo `main` that uses the class `EncryptedText`:

```
    public class ClientEncryptedText {
      public static void main(String[] args) {
        EncryptedText t = new EncryptedText("Nel mezzo del cammin di ...", 10);
        System.out.println(t.getEncryptedText());
        System.out.println(t.getDecryptedText(10));
        t.setKey(10,20);
        System.out.println(t.getDecryptedText(10));
        System.out.println(t.getDecryptedText(20));
      }
    }
```

The output of the program is the following:

```
Xov*wo??y*nov*mkwwsx*ns*888
Nel mezzo del cammin di ...
null
Nel mezzo del cammin di ...
```

## 6.55  The class `StringTokenizer`

The class `StringTokenizer` allows us to divide a string into tokens. A *token* is the maximal sequence of consecutive characters of a string that are not delimiters. The default delimiters are " \t\n\r\f", i.e., the space character, the tab character, the newline character, the return character, and the form-feed character.

*Example:* The tokens of the string:

```
"I am    a
  student
of Introduction     to Programming"
```

are: "I", "am", "a", "student", "of", "Introduction", "to", "Programming".

An object of type `StringTokenizer` is constructed starting from a string to be tokenized, and internally maintains a current position within the string. The class exports some methods that allow us to advance this position, and to return the token as a substring of the string used to create the `StringTokenizer` object.

The `StringTokenizer` class has the following constructors (among others):

- `StringTokenizer(String str)`
  Constructs a string tokenizer for the specified string, using the default delimiter set " \t\n\r\f".
- `StringTokenizer(String str, String delim)`
  Constructs a string tokenizer for the specified string, using a specified set of delimiters.

The `StringTokenizer` class has the following methods (among others):

- `boolean hasMoreTokens()`
  Tests if there are more tokens available from this tokenizer's string.
- `String nextToken()`
  Returns the next token from this string tokenizer.

*Example:* Use of an object of type `StringTokenizer`. The following fragment of code

```
StringTokenizer st = new StringTokenizer("I am   a   \n  student");
while (st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

prints the following output:

```
I
am
a
student
```

*Notes:*

- The class `StringTokenizer` is part of the package `java.util`. Hence, each time we want to use this class, we have to import it explicitly through the statement `import java.util.StringTokenizer;`.
- In fact, the class `StringTokenizer` is outdated and is kept only for compatibility reasons. Instead the `split` method of the `String` class or of the `Pattern` class in the `java.util.regex` package should be used. However, these methods make use of arrays, which we will see only later.

## Exercises

**Exercise 6.1.** Write a program that reads 10 (arbitrary) integers and prints the smallest one.

**Exercise 6.2.** Write a public static method that takes as parameter a positive integer $n$, and prints the first $n$ even numbers.

**Exercise 6.3.** Write a public static method that takes as parameter a positive integer $n$, and computes and returns the factorial of $n$. The method should also print a suitable message when $n$ is negative.

**Exercise 6.4.** Write a program that reads from input an integer $n$ and a sequence of integers of length $n$, and prints the sum of the positive and the sum of the negative integers in the sequence.

**Exercise 6.5.**    The value of $\pi$ can be calculated with the series

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \cdots$$

Write a public static method that takes as parameter an integer $n$, and computes and returns the value of $\pi$ approximated to the first $n$ terms of the series.

**Exercise 6.6.**    Write a public static method that takes as parameters a string and an integer $d$, and returns the string suitably decoded according to $d$. The decoded string is obtained by replacing each character $c$ in the string with the character that has code equal to the code of $c$ decremented by $d$.

**Exercise 6.7.**    Write a public static method that takes as parameters a string and a character $c$, and returns the position of the first character of the longest sequence of consecutive $c$'s in the string. If $c$ does not occur at all in the string, the method should return -1.

**Exercise 6.8.**    Modify the program for printing the multiplication table in such a way that the printed numbers are aligned in columns.

**Exercise 6.9.**    Write a public static method that takes and parameter an integer $h$ between 1 and 9, and prints a pyramid of numbers of height $h$.
*Example:* For $h = 4$ the method should print the pyramid

```
   1
  121
 12321
1234321
```

**Exercise 6.10.**    Write a program that reads from input and integer $n$ and prints the factorial of all numbers between 1 and $n$. Make use of the method defined in Exercise 6.3.

**Exercise 6.11.**    A positive integer is said to be *prime* if it is divisible only by 1 and by itself. Write a public static method that takes as parameter a positive integer, and returns a boolean that indicates whether the integer is prime or not.

**Exercise 6.12.**    Write a program that reads from input an integer $n$, and prints all prime numbers between 2 and $n$. Make use of the method defined in Exercise 6.11.

**Exercise 6.13.**    Write a program that reads from input an integer $n$, and prints the first $n$ prime numbers (by convention, 1 is not considered to be prime). Make use of the method defined in Exercise 6.11.

**Exercise 6.14.**    Write a program that reads from input an integer $n$, and prints all its prime factors. For example, if the integer is 220, the program should print: `2, 2, 5, 11`. Make use of the method defined in Exercise 6.11.

**Exercise 6.15.**    Realize a Java class to represent messages. Each message is characterized by:

- a sender,
- a receiver,
- the text of the message.

All three types of information should be represented as strings. Besides the functionalities of getting and setting the sender, the receiver, and the text, messages should support the operation of swapping the sender and the receiver, and three forms of compression:

1. elimination of the white spaces at the beginning and at the end of the text, and replacement of each sequence of more than one whitespace (used to separate words) with a single white space;
2. elimination of all vowels;
3. both (1) and (2).