Unit 10

Program errors and exception handling

Summary

- Types of program errors
- Syntax, semantic, and logical errors
- Compile time and runtime errors
- Test drivers
- Debugging techniques
- Exception handling
- The most common types of exceptions
- The `throws` clause and the `throw` statement
- Catching exceptions by means of the `try-catch` construct
- Propagation of exceptions
- Exceptions when reading from a file

10.1 Types of program errors

We distinguish between the following types of errors:

1. **Syntax errors**: errors due to the fact that the syntax of the language is not respected.
2. **Semantic errors**: errors due to an improper use of program statements.
3. **Logical errors**: errors due to the fact that the specification is not respected.

From the point of view of when errors are detected, we distinguish:

1. **Compile time errors**: syntax errors and static semantic errors indicated by the compiler.
2. **Runtime errors**: dynamic semantic errors, and logical errors, that cannot be detected by the compiler (debugging).

10.2 Syntax errors

Syntax errors are due to the fact that the syntax of the Java language is not respected.

Let us see some examples of syntax errors.

Example 1: Missing semicolon:

```
int a = 5 // semicolon is missing
```

Compiler message:

```
Example.java:20: ';' expected
int a = 5
```

Example 2: Errors in expressions:

```
x = ( 3 + 5;  // missing closing parenthesis ')'
y = 3 + * 5;   // missing argument between '+' and '*'
```

10.3 Semantic errors

Semantic errors indicate an improper use of Java statements.

Let us see some examples of semantic errors.

Example 1: Use of a non-initialized variable:

```
int i;
i++;    // the variable i is not initialized
```

Example 2: Type incompatibility:

```
int a = "hello"; // the types String and int are not compatible
```

Example 3: Errors in expressions:

```
String s = "...";
int a = 5 - s;  // the - operator does not support arguments of type String
```

Example 4: Unknown references:

```
Strin x;                      // Strin is not defined
system.out.println("hello"); // system is not defined
String s;
s.println();        // println is not a method of the class String
```

Example 5: Array index out of range (dynamic semantic error)

```
int[] v = new int[10];
v[10] = 100;           // 10 is not a legal index for an array of 10 elements
```

The array `v` has been created with 10 elements (with indexes ranging from 0 to 9), and we are trying to access the element with index 10, which does not exist. This type of error is not caught during compilation, but causes an exception to be thrown at runtime.

10.4 Logical errors

Logical errors are caused by the fact that the software specification is not respected. The program is compiled and executed without errors, but does not generate the requested result.

Let us see some examples of logical errors:

Example 1: Errors in the performed computation:

```
public static int sum(int a, int b) {
  return a - b ;
}
// this method returns the wrong value wrt the specification that requires
// to sum two integers
```

Example 2: Non termination:

```
String s = br.readLine();
while (s != null) {
  System.out.println(s);
}   // this loop does not terminate
```

10.5 Errors detected by the compiler and runtime errors

All syntax errors and some of the semantic errors (the static semantic errors) are detected by the compiler, which generates a message indicating the type of error and the position in the Java source file where the error occurred (notice that the actual error could have occurred before the position signaled by the compiler).

Other semantic errors (the dynamic semantic errors) and the logical errors cannot be detected by the compiler, and hence they are detected only when the program is executed.

Let us see some examples of errors detected at runtime:

Example 1: Division by zero:

```
int a, b, x;
a = 10;
b = Integer.parseInt(kb.readLine());
x = a / b;  //ERROR if b = 0
```

This error occurs only for a certain configuration of the input (`b = 0`).

Example 2: File does not exist:

```
FileReader f = new FileReader("pippo.txt");
```

The error occurs only if the file `pippo.txt` does not exist on the harddisk.

Example 3: Dereferencing of a `null` reference:

```
String s, t;
s = null;
t = s.concat("a");
```

The `concat()` method cannot be applied to a reference whose value is `null`. Note that the above code is syntactically correct, since the `concat()` method is correctly applied to a reference of type `String`, but it contains a dynamic semantic error due the fact that a method cannot be applied to a reference whose value is `null`.

10.6 Drivers for testing

Drivers for testing are program portions that are used to test the correctness of a class or of a method. The purpose of such drivers is to call all methods of the public interface of a class and verify that they respect the specification.

In order to perform a test that is complete, we should follow some guidelines:

- verify each functionality (each method);
- perform the tests according to a specific order (the order of method application is often important);
- ensure that each statement is executed at least once (for example, when we have a conditional statement, we have to perform the test for various configurations of the input, in such a way that the boolean condition becomes respectively true and false);
- detect and test special cases (for example, an empty file as input to a method that reads from a file).

10.7 Techniques for detecting errors (debugging)

If the testing phase signals the presence of logical errors, or if we are not able to detect the cause for a runtime error, it is necessary to **debug** the program.

There are two ways in which we can obtain information that is helpful for debugging a program:

1. by inserting output statements in the code;
2. by executing the program by means of a **debugger**.

10.8 Debugging by inserting output statements

This debugging technique is based on inserting in suitable positions of the source code statements that print the content of variables that could contain wrong values causing an error.

Example:

```
int a, b, x;
a = 5;
b = Integer.parseInt(kb.readLine()); // reading of b
...         // statements that do not change b
x = a/b;
```

To *debug* this program statement we can verify, by printing the value of `b` on the screen, that the error occurs when the variable `b` has value 0.

```
int a, b, x;
a = 5;
b = Integer.parseInt(kb.readLine()); // reading of b
...         // statements that do not change b
System.out.println("b = " + b);
x = a/b;
```

Once the causes for the error have been identified and corrected, the print statements can be removed before closing the debugging session.

Note: If it is necessary to explore the content of objects, rather than of simple variables of primitive data types, we can make use of the `toString()` method, which provides information on the content of the object. We could also redefine `toString()` so as to simplify the reading of the state of the object during debugging.

10.9 Execution of the program by means of a debugger

A debugger allows us to:

- execute the statements of a program one at a time,
- execute the program until the execution reaches certain points defined by the user (called **breakpoints**),
- examine the content of variables and objects at any time during the execution.

Debuggers are very useful tools to detect the causes of errors in programs.

Example 1:

```
int a, b, x;
a = 5;
b = Integer.parseInt(kb.readLine()); // reading of b
...         // statements that do not change b
x = a/b;
```

By means of a debugger we can verify the value of the variable `b` before executing the statement that generates the error.

Example 2:

```
String s, t;
s = null;
...
t = s.concat("a");
```

The assignment statement for `t` generates an exception of type `NullPointerException`. Such an error depends on the fact that, when the assignment statement is executed, the value of `s` is `null`. To check this error, we can use a debugger and observe the value of the variable `s` before executing the statement that generates the error.

10.10 Handling errors during the execution of programs

During the execution of a program, various conditions can occur that cause an unexpected and abnormal termination of the program.

Example: Consider the following program:

```
public class TestException {
  public static void main (String[] args) {
    int falseNumber = Integer.parseInt("OK");
    System.out.println("this println statement is not executed");
  }
}
```

the following message is printed on the screen:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "OK"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:468)
```

```
  at java.lang.Integer.parseInt(Integer.java:518)
  at TestException.main(TestException.java:3)
```

Hence, the string `"this println statement is not executed"` is not printed.

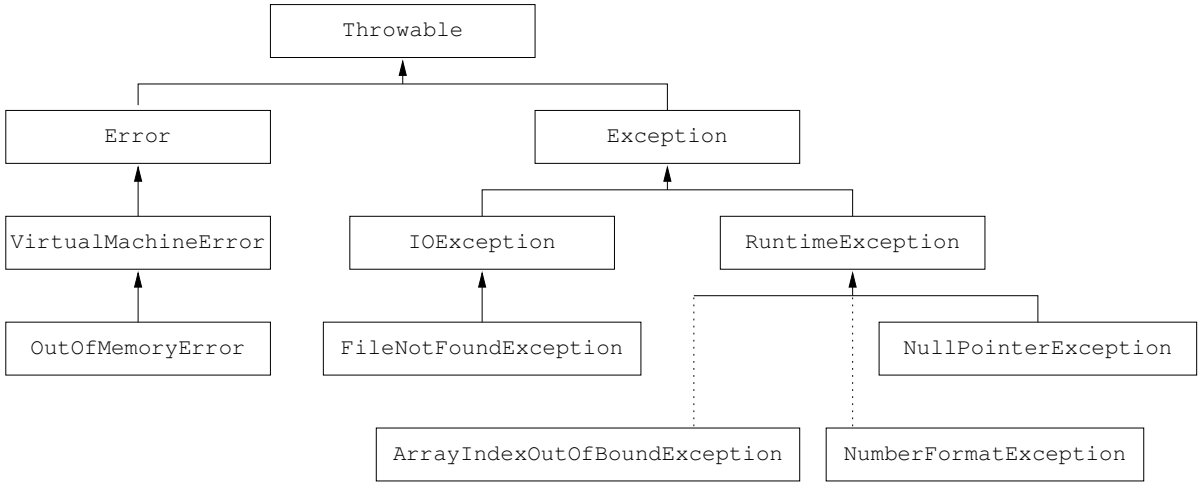
In Java, the errors that occur at runtime are represented by means of **exceptions**. Java offers a predefined set of exceptions that can be thrown during program execution.

To avoid that a program terminates unexpectedly, Java allows us to handle exceptions by means of a suitable construct.

10.11 The hierarchy of exceptions

Exceptions and errors are represented through Java classes and they are organized in a hierarchy.

The class `Throwable` is the superclass of all types of errors and of all exceptions. Errors represent events that cannot be controlled by the programmer (for example, `OutOfMemoryError`), while exceptions can be handled during the execution of the program.



Note: In this figure, dashed lines represent an indirect inheritance relation.

10.12 How to handle exceptions

There are two possibilities for handling exceptions:

- handling the exception where it is generated
- handling the exception in another point of the program

If the exception is not handled, instead, the program terminates and prints out a message.

For example, if we try to do a division by zero, we obtain:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at DivByZero.main(DivByZero.java:7)
```

10.13 The most common types of exceptions

1. `NullPointerException`: generated when the reference used to invoke a method has value `null`, or when we try to access an instance variable through a `null` reference. Some methods throw explicitly this type of exception when they are passed a parameter that is `null`.
2. `ArrayIndexOutOfBoundsException`: generated when we access an element of an array using an index that is less than zero, or bigger than the length of the array minus one.
3. `IOException`: generated by methods that access input/output devices when an error situation occurs.
4. `FileNotFoundException`: generated when we try to open a non-existent file.
5. `NumberFormatException`: generated by methods that perform conversions from strings to numbers. For example, `Integer.parseInt()` generates an exception of this type if the string passed as parameter does not contain a number.

10.14 The `throws` clause

All Java methods can use the `throws` clause to handle the exceptions generated by the methods they invoke.

The `throws` clause is added to the header of the method definition. For example:

```
public static void main(String[] args) throws IOException {
  ...
}
```

10.15 Checked exceptions and runtime exceptions

Checked exceptions must be mentioned in the `throws` clause of all methods in which they could occur. Such exceptions are then propagated to the methods that call the one in which the exception occurs.

For example, if a method `A` declares to throw an exception of type `MyException`, all methods that call `A` must either declare themselves that they throw such an exception, or catch the exception (see later). Hence, the following code fragment is wrong, because the method `B` does not declare to throw exceptions of type `MyException` (and does not catch the exception that could be generated by `A`).

```
public static void A() throws MyException {
  ...
}

public static void B() {
  A();
}
```

Unchecked exceptions are objects of type (subclass of) `RuntimeException`. They represent exceptions that occur in the Java virtual machine during program execution. This class includes:

- arithmetic exceptions (example: division by zero),
- reference exceptions (example: try to access an object through a reference that is `null`), and
- exceptions when indexing an array (example: trying to access an element of the array with an index that is either too small or too large).

Note: It is not necessary that the subclasses of `RuntimeException` are mentioned in the `throws` clause (this is why they are celled *unchecked*).

10.16 Definition of new exceptions

A new exception can be defined starting from the class `Exception` or from one of its descendants.

```
public class MyException extends Exception {
  public MyException (String message) {
    super(message);
  }
}
```

The class `MyException` specifies a particular message to visualize. The constructor of an `Exception` takes as parameter a `String` that is printed when the exception occurs.

It is also possible to define an exception class as a subclass of `RuntimeException`, instead of `Exception`. In this way, it will not be necessary to handle explicitly such exceptions, since they are of type *unchecked*.

10.17 The `throw` statement

To throw an exception, we use the `throw` statement.

Throwing an exception

Syntax:

`throw` *`exceptionObject`* `;`

- *`exceptionObject`* is an object of the class `Exception` (or of one of its subclasses)

Semantics:

Throws an exception of the type indicated by the object appearing as parameter to the `throw` statement.

Example:

```
throw new MyException("message for MyException");
```

This statement throws an exception of type `MyException`, which prints on the screen the message that appears as parameter to the constructor.

10.18 Example

```
import java.io.*;

public class UseMyException {
  public static void main(String[] args) throws MyException, IOException {
    int min = 10, max = 30;
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Input a number between " + min +
                       " and " + max + " included");
    int value = Integer.parseInt(br.readLine());
    if (value < min || value > max)
       throw new MyException("The value is not in the allowed interval");
    System.out.println("The value is in the allowed interval");
  }
}
```

The following is an example of a run of the program:

```
Input a number between 10 and 30 included
9
Exception in thread "main" MyException: The value is not in the allowed interval
  at UseMyException.main(UseMyException.java:11)
```

Note: If we had defined the class `MyException` as a subclass of `RuntimeException`, instead of `Exception`, then we could have avoided to indicate `MyException` in the `throws` clause of the definition of the `main()` method.

10.19 How to catch an exception

The `throw` statement starts a chain of method terminations that starts with the method that executes the `throw` statement, and goes up to the calling methods till the `main()` method.

If an exception is *caught*, the chain of exceptions that would lead to the termination of the program is interrupted.

To catch an exception, we can use the `try-catch-finally` construct.

The `try-catch-finally` construct

Syntax:

```
try {
  try-block
}
catch(ExceptionClass1 e) {
  catch-block
}
catch(ExceptionClass2 e) {
  catch-block
}
...
finally {
  finally-block
```

```
}
```

- *try-block* : sequence of statements that will be executed under the control of the following `catch` clauses
- *catch-block* : sequence of statements that will be executed if a statement in the *try-block* generates an exception of the type specified in the corresponding `catch` clause
- *finally-block* : sequence of statements that will be always executed (both in the case where the *try-block* is executed without exceptions, and in the case where a *catch-block* is executed to catch an exception).

Semantics:

Catches one or more exceptions that can occur in a code fragment. The execution of the statements in the *try-block* is interrupted in the case where one of these statements generates an exception. If this happens, the `catch` clauses are evaluated in the order in which they are written, and the *catch-block* is executed that corresponds to the first clause for which the generated exception belongs to the specified class. Finally, the statements in the *finally-block* are executed.

If no statement in the *try-block* generates an exception, then, at the end of its execution, only the statements in *finally-block* are executed.

Example:

```
try {
   System.out.println(Integer.parseInt(br.readLine());
}
catch(IOException e1) {
   System.out.println("An IO error occurred.");
}
catch(NumberFormatException e2) {
   System.out.println("The string read does not contain an integer.");
}
finally {
   System.out.println("Block executed.");
}
```

This code fragment tries to convert a string read from an input channel to an integer and to print the integer. If an IO error occurs, or if the string read does not contain a sequence of digits, a corresponding error message is printed on the video. In any case, the `println()` statement for the string `"Block executed"`, corresponding to the `finally` clause, is executed.

10.20 The `getMessage()` method

The class `Exception`, and hence also all its subclasses, have a method `getMessage()`, which allows one to extract the string associated to the exception.

Example:

```
try {
  ...
}
catch (NumberFormatException e) {
  System.out.println("Catched NumberFormatException");
  System.out.println(e.getMessage());
}
catch (IOException e) {
  System.out.println("Catched IOException");
  System.out.println(e.getMessage());
}
```

10.21 Example of exception handling

Write a Java program that prints the maximum of the sequence of non negative integer values that are stored on the file `data.txt`.

We first concentrate on the problem without considering exceptions.

```
import java.io.*;

public class MaximumWithoutExceptions {
  public static void main (String args[]) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader("data.txt"));
            // could generate FileNotFoundException (checked)
    int max = -1;

    String line = br.readLine();
            // could generate IOException (checked)
    while (line != null) {
      int n = Integer.parseInt(line);
          // could generate NumberFormatException (unchecked)
      if (n > max) max = n;
      line = br.readLine();
          // could generate IOException (checked)
    }
    System.out.println("Maximum = " + max);
  }
}
```

Let us now consider exceptions.

```
import java.io.*;

public class MaximumWithExceptions {
  public static void main (String args[]) {
    try {
      BufferedReader br = new BufferedReader(new FileReader("data.txt"));
                // could generate FileNotFoundException (checked)
      int max = -1;

      String line = br.readLine();
                // could generate IOException (checked)
      while (line != null) {
        int n = Integer.parseInt(line);
                // could generate NumberFormatException (unchecked)
        if (n > max) max = n;
        line = br.readLine();
                // could generate IOException (checked)
      }
      if (max == -1)
         throw new Exception("File empty or all numbers < 0");
      else
         System.out.println("Maximum = " + max);
      }
    catch (FileNotFoundException e) {
       System.out.println("The file does not exist.");
    }
    catch (IOException e) {
       System.out.println("The file cannot be read.");
    }
```

```
      catch (NumberFormatException e) {
         System.out.println("The file contains non numeric data.");
      }
      catch (Exception e) {
         System.out.println(e.getMessage());
      }
    }
  }
```

If the file contains alphanumeric data that cannot be converted to integer values, the first program would generate the following error message:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "pippo"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
  at java.lang.Integer.parseInt(Integer.java:468)
  at java.lang.Integer.parseInt(Integer.java:518)
  at MaximumWithoutExceptions.main(MaximumWithoutExceptions.java:12)
```

The second program, instead, would handle the exception and print:

```
The file contains non numeric data.
```

10.22 Propagation of exceptions

If an exception is not caught and handled where it is thrown, the control is passed to the method that has invoked the method where the exception was thrown. The propagation continues until the exception is caught, or the control passes to the `main()` method, which terminates the program and produces an error message.

It is the `throw` statements that starts the chain of method terminations. For example:

```
import java.io.*;

public class ExceptionPropagation1 {

  public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Insert a number:");
    int c = Integer.parseInt(br.readLine());
    first(c);
  }

  private static void first(int a) throws Exception {
    second(a);
  }

  private static void second(int b) throws Exception {
    Exception propagate = new Exception("The value is too small.");
    if (b < 10)
      throw propagate;
    System.out.println("OK");
  }
}
```

If the program reads a value less than 10, for example 5, an exception is thrown and the following messages are printed:

```
Insert a number:
5
Exception in thread "main" java.lang.Exception: The value is too small.
  at ExceptionPropagation1.second(ExceptionPropagation1.java:17)
  at ExceptionPropagation1.first(ExceptionPropagation1.java:13)
  at ExceptionPropagation1.main(ExceptionPropagation1.java:9)
```

Note that, in order to allow the chain of terminations started by the `second()` method to propagate to the `main()` method, it is necessary that all methods that are part of the termination chain have in their header the `throws` clause with the list of involved exceptions: in this case, only `Exception`.

10.23 Interrupting the propagation of exceptions

In the following example, the exception is not handled in the method that generates it, but in a method that invokes the method that generates the exception. Then, the exception is not further propagated upwards. Hence, the `first()` and `main()` methods do not have to declare in the `throws` clause that they throw the exception.

```
import java.io.*;

public class ExceptionPropagation2 {

  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Insert a number:");
    int c = Integer.parseInt(br.readLine());
    first(c);
  }

  private static void first(int a) {
    try {
      second(a);
    }
    catch (Exception e) {
      System.out.println("Exception handled in the first method.");
      System.out.println(e.getMessage());
    }
  }

  private static void second(int b) throws Exception {
    Exception propagate = new Exception("The value is too small.");
    if (b < 10)
      throw propagate;
    System.out.println("OK");
  }
}
```

The result of the execution of this program for the same input as before, is the following:

```
Insert a number:
5
Exception handled in the first method.
The value is too small.
```

10.24 Example of exception handling when reading from a file

Write a method `printIntegers()` that reads integer values from a file and prints them to standard output. For those lines of the file that do not contain an integer value (e.g., they contain alphabetic characters) the program should print the string `"***"` and continue reading from the file.

```
public static void printIntegers(String filename) {
  try {  // cattura exceptions di IO
    FileReader f = new FileReader(filename);
    BufferedReader in = new BufferedReader(f);
    String line = in.readLine();
    while (line != null) {
      try { // capture NumberFormaException
```

```
          int n = Integer.parseInt(line);
          System.out.println(n);
        }
        catch(NumberFormatException e) {
          System.out.println("***");
        }
        line = in.readLine();
     }
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
```

Exercises

Exercise 10.1. Determine whether the following program will generate *(i)* compilation errors, *(ii)* runtime errors. If the program does not generate errors, say what it will print out; if the program generates errors, correct them and say what it will print out after the correction. Motivate your answers.

```
public class Exercise1 {
  public static void main(String[] args) {
    for (int i = 0, j = 0; i < 10, j < 10; i++, j++) {
      System.out.println(i + " + " + j + " = " + (i+j));
    }
    System.out.println("I've printed out the sums of i and j up to "
                       + i + "," + j);
  }
}
```

Exercise 10.2. Determine whether the following program will generate *(i)* compilation errors, *(ii)* runtime errors. If the program does not generate errors, say what it will print out; if the program generates errors, correct them and say what it will print out after the correction. Motivate your answers.

```
public class Exercise2 {

  private int x = 101;

  private void f(int x) {
    x++;
    g();
  }

  private void g() {
    System.out.println(x);
  }

  public static void main(String[] args) {
    Exercise2 e = new Exercise2();
    int x = 200;
    e.f(x);
  }
}
```

Exercise 10.3. Determine whether the following classes will generate *(i)* compilation errors, *(ii)* runtime errors. If the program does not generate errors, say what it will print out; if the program generates errors, correct them and say what it will print out after the correction. Motivate your answers.

```
public class Base {
  public Base() {
    infob = "I am an object of the Base class";
  }
  public String getInfo() {
    return infob;
  }
  private String infob;
}

public class Derived extends Base {
  public Derived() {
    super();
    infod = "I am an object of the Derived class";
  }
  public String getInfo() {
    return infod + ", " + super.getInfo();
  }
  private String infod;
}

public class Exercise3 {
  public static void main(String[] args) {
    Base b = new Base();
    Derived d = new Derived();
    System.out.println(b.getInfo());
    System.out.println(d.getInfo());
    b = d;
    System.out.println(b.getInfo());
  }
}
```

Exercise 10.4. Capture all exceptions in the following program, printing out error messages that describe the type of error that occurred.

```
import java.io.*;

public class Exercise4 {

  public static void main(String[] args) {
    int n=10;
    int[] v = new int[n];
    FileReader f = new FileReader("dati.txt");
    BufferedReader in = new BufferedReader(f);
    int i=0;
    String linea = in.readLine();
    while (linea!=null) {
      v[i] = Integer.parseInt(linea);
      linea = in.readLine();
      i++;
    }
    f.close();
  }
}
```

Exercise 10.5. Solve Exercise 9.6 by handling explicitly all exceptions by printing out suitable error messages.

Exercise 10.6. Define a new exception, called `ExceptionLineTooLong`, that prints out the error message `"The strings is too long"`. Write a program that reads all lines of a file and throws an exception of type `ExceptionLineTooLong` in the case where a string of the file is longer than 80 characters. Handle also all exceptions that could be thrown by the program.

Exercise 10.7. Write a class containing the following static methods:

- `fileExists()`, that takes as a parameter a string and returns a boolean value that is `true`, if the file whose name is passed as parameter exists, and `false` otherwise;
- `isInt()`, that takes as parameter a string and returns a boolean value that is `true` if the string represents an integer, and `false` otherwise.
- `isDouble()`, that takes as parameter a string and returns a boolean value that is `true` if the string represents a real number, and `false` otherwise.

Solve the exercise by suitably catching exceptions.

Exercise 10.8. Define the exceptions that are necessary to catch the possible errors that can occur in the class `Matrix` of Exercise 9.9.

- `ExceptionWrongMatrixValues` that is thrown in the method `read()` if the data on the file does not correspond to numeric values, or if the data are not consistent with the form of a matrix (e.g., the rows have different length);
- `ExceptionWrongMatrixDimension` that is thrown in the method `read()` if the data on the file do not correspond to the dimension of the matrix.

Modify the class `Matrix` in such a way that it generates the new exceptions when necessary.