PRG (E.T.S. d'Enginyeria Informàtica)
Academic Year 2019-2020

# *Lab activity 4. Exception management and files*
# *First part: A utility library for reading from standard input*

(One session)

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

## Contents



# 1 Context and objectives

In the academic framework, this activity is related to "*Unit 3: elements of OOP: inheritance and exception handling*" and to "*Unit 4: I/O: files and streams*". The main objective of this activity is to reinforce and use the concepts that were introduced in the theoretical lectures on exception handling and I/O management by using files and streams. More specifically:

- Throw, ignore, and catch exceptions, both locally and remotely
- Read/write from/to a text file
- Handle I/O exceptions

In order to do that, in this lab activity it is proposed to develop a small application in which data items which are read from text files are processed, storing the result into another file.

The activities are organised in two parts. In this first part, it is proposed to develop a small utility library that allows reading numerical data from the standard input easily, while in the second part the different parts of the application class would be completed.

# 2 The `utilPRG` library

The utility library `utilPRG` would consist of a package that contains the `CorrectReading` class, whose methods would be completed in this part of the activity.

## 2.1 Activity 1: set up `pract4` package

- If you develop the activity in your own computer, create the `prg` project into the folder you prefer. If you develop the activity by using the remote desktop, the `prg` project would be the one located in `DiscoW`
- Open with *BlueJ* the `prg` project and create a package `pract4` for this activity. Get inside the `pract4` package and create a subpackage `utilPRG`
- Download from `Recursos/Laboratorio/Práctica 4/English/code` from *Poliformat* of `PRG` the files `CorrectReading.java` and `TestCorrectReading.class`, and move them into the subpackage `utilPRG` (remember that the `.class` file cannot be added from BlueJ and that you must copy it to the disk folder that corresponds to the subpackage)

In Figure 1 you can see the windows that correspond to the `prg` project and the packages that you have created.

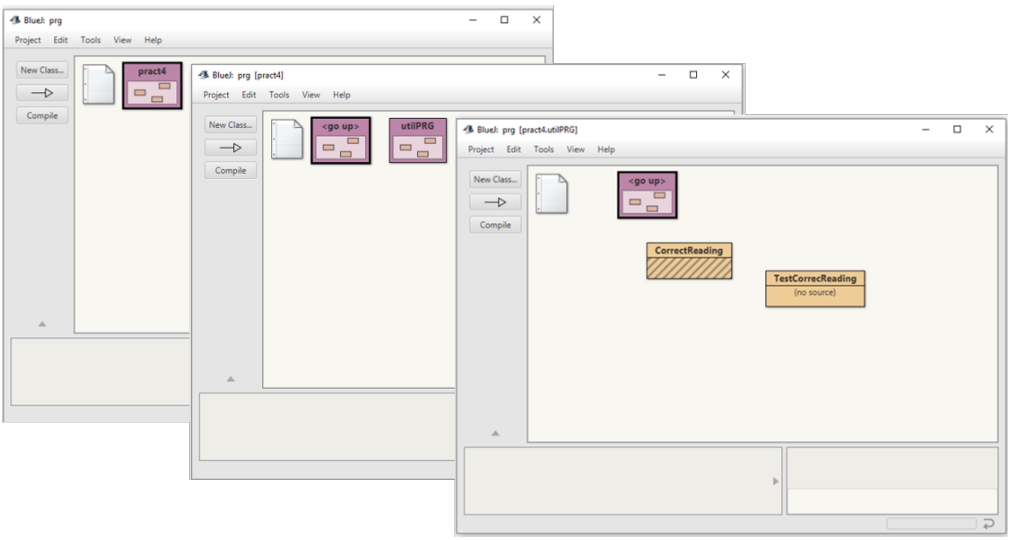


Figure 1: Project `prg`, package `prg[pract4]`, and subpackage `prg[pract4.utilPRG]`.

# 3 Error detection from keyboard input

The `nextInt()` method from `Scanner` has been used frequently. In the documentation (`http://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html`) it can be seen that when the value input by the user is not an integer, it raises the `InputMismatch Exception` exception, whose documentation can be accessed as well.

This exception is an **unchecked** exception (derivated from `RuntimeException`), and consequently it is situated in the class hierarchy in the place that is shown in Figure 2. Java does not ask for mandatory exception handling for this class exceptions, although it can be useful to catch and handle them when they appear, as in the methods proposed in the following activities.

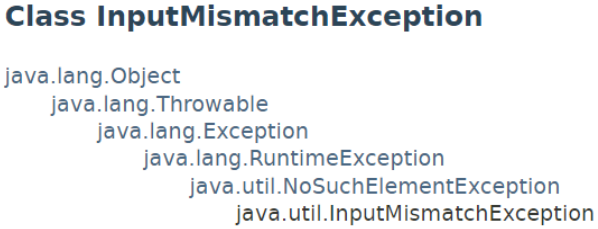


Figure 2: Class hierarchy for `InputMismatchException` exception.

## 3.1 Activity 2: test and check `nextInt(Scanner, String)`

The method `nextInt(Scanner, String)` from the class `CorrectReading` allows to read an `int` value from keyboard by using the method `nextInt()` from the `Scanner` class, but capturing the exception in case it raises, and showing an error message to show the user the possible actions to correct the error.

This method can be tested in the *BlueJ* Code Pad. For that, execute the following statements:

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
int value = CorrectReading.nextInt(sc, "Value: ");
```

From the terminal window of *BlueJ*, input a non-integer value (e.g., the character of the word "hello") and press *Enter* to send the typed characters to the program (including the newline character). You can see the message that indicates that the input value is not valid and that the method will run until an integer value is given.

Let's check the code of the method, where you can see that it employs a `try-catch-finally` method to handle the exception: the method repeats reading operations until an integer value is read.

Pay careful attention to the `finally` statement of the `nextInt(Scanner, String)` method, which is used to "clean" the input buffer. If the buffer is not cleaned, the method would run incorrectly.

For example, suppose that initially we input from the keyboard the character sequence `23`, and then *Enter* is pressed. Then, `sc.nextInt()` returns the `int` value 23, and our method finishes returning that value, but the newline character (`'\n'`) remains in the buffer. If then another part of the code asks the user for a new line, read by the method `nextLine()` and with the same `Scanner` object, it will return the empty `String` (`""`) since it reads the newline from the buffer, without allowing the user to type anything.

Suppose now that the `hello` character sequence is typed and the *Enter* key is pressed. In that case, `nextInt()` raises the `InputMismatchException` exception without extracting any character from the buffer. Thus, the token that would be available for the next loop iteration is the same.

As we know, a `finally` statement is executed independently on the level of execution of the instructions in the `try` block (and no `catch` block is executed) or when an exception raises and a `catch` block is executed. Thus, here `finally` was used to execute the `sc.nextLine()` instruction in any input case:

- the token that was read with `sc.nextInt()` is correct and the `try` block finishes completely. Then, `sc.nextLine()` allows to clear the newline character created by the key *Enter* pressed by the user

- the token was incorrect, `sc.nextInt()` fails and it does not extract the token from the buffer, and the exception is raised (the `catch` block shows the corresponding warning). Then, `sc.nextLine()` allows to extract that token from the buffer, including the newline character; otherwise, next attemps would find the same problem repeatedly, creating an infinite loop

The class implements a similar method `nextDouble(Scanner, String)` for handling the `InputMismatchException` exceptions that can be raised by `sc.nextDouble()`.

## 3.2 Activity 3: complete `nextDoublePositive(Scanner, String)`

In the class `CorrectReading`, the method `nextDoublePositive(Scanner, String)` must capture the `InputMismatchException` exception when the value typed by the used is not a `double`, in a similar way to that employed for the `nextInt(Scanner, String)` method, and showing a proper error message instead of terminating the execution. In the documentation (comments) of the method you can find examples of the messages that must be shown.

Complete the method by adding to it the needed `try-catch-finally` block. In that way, you would have added an exception handler for detecting and handling an exception at local level, that is, in the same method where the exception is raised.

To test this method, execute in the Code Pad the following instructions; in order to read real numbers with the usual `double` notation, the `Scanner sc` object used for the tests must be configured with `Locale.US`, as the instructions show:

```
import java.util.Scanner;
import java.util.Locale;
Scanner sc = new Scanner(System.in).useLocale(Locale.US);
double value = CorrectReading.nextDoublePositive(sc, "Value: ");
```

## 3.3 Activity 4: complete `nextInt(Scanner, String, int, int)`

- In the class `CorrectReading`, the method `nextInt(Scanner, String, int, int)` must be completed in order to capture the `InputMismatchException` when the value typed by the user is not an `int`, in a way similar to that employed for the `nextInt(Scanner, String)` method, showing a proper error message instead of terminating the execution
- Moreover, the same method must control that the typed value is in the range `[lowerBound, upperBound]`. You have two basic ways for doing that:
  - add a proper condition in the loop guard, or
  - throw an exception

  In this method you must employ this last option; for that, by using the `throw` statement, you must add a conditional instruction such that when then typed value is not in the given range, the `IllegalArgumentException` exception is raised, along with a message that indicates that the read value is not in the given range

  After that, you can add a new `catch` block to capture locally that exception, in a similar way to what you did with `InputMismatchException`, showing the exception message by using the `getMessage()` method (which is inherited from the `Throwable` class)

## 3.4 Activity 5: testing the methods. `TestCorrectReading` class

The methods of the `CorrectReading` library can be tested by using the given `TestCorrectReading` class. However, this test **does not** give information on where the error is, but only if all the code is correct or not. To find the error you must do the proper tests, by writing and executing a test program or by using the Code Pad.