PRG – (E.T.S. de Ingeniería Informática) – Academic year 2017-2018
Lab practice 4 – Files and exception handling
First and second of three sessions

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Contents

1 Context and previous work **1**

2 Problem statement **2**

3 Checked and unchecked exceptions **3**

4 Detection of runtime errors **4**

5 Catching exceptions. A first example **4**

6 Explicitly ignoring exceptions vs locally handling them **6**

7 Evaluation **10**

1 Context and previous work

This lab practice is about units 3 and 4:

Unit 3: *Elements of Object Oriented Programming: inheritance and exception handling*

Unit 4: *Input/Output: Files and Streams*

The main goal to be reached is that students become familiar with exception handling and management of input/output streams/files. In particular:

- Throw, ignore, and catch exceptions, both locally and remotely.
- Read from (or write to) text files.
- Handle exceptions related to input/output.

For it, during the three sessions of this lab practice, we are going to develop a small application to process data loaded from text files and saving the results into another file.

2 Problem statement

There is available the register of the number of accidents during one year. As the register is the result of merging several registers from different cities or regions, it will be normal to work with several text files, but all of them with the same format. Each line in those files contains three integer values:

```
day  month  count
```

where `count` is the value registered in a given day of the year: `day` and `month`.

The same date can appear more than once, even in the same file, because each file could be the concatenation of several files, each one corresponding to a different city or region. So, the dates do not appear in chronological order.

The goal is to have an application that extracts all the data of a year from a set of text files, and creates a new text file with the results, then the results will appear in chronological order. Obviously, each date must appear once in the result, so the application must accumulate the counts of each date.

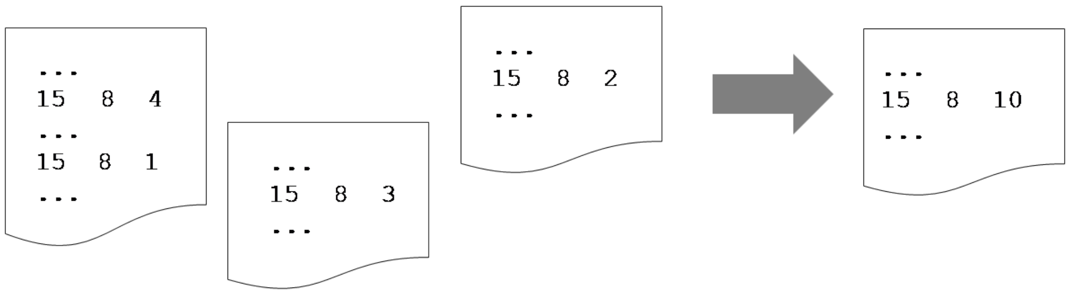

Figure 1: Data aggregation from one or several text files.

The following classes are provided as the starting point for developing the application:

- Class `SortedRegister` for facing the problem by means of two dimensional array where the month will be the index for rows, and the day the index for columns.

  Thanks to that, the count of each line from text files will go to the element of the two dimensional array indexed by the month and the day. Aggregating the values is, then, so simple.

  Figure 2 show the data structure of the objects of this class. Realise that elements at position 0 of each row or column are not used, thanks to that, the values of month and day are used as indexes with no additional operations.

  Accessing all the valid elements in the proper order, row-by-row then column-by-column, will give us the chronological order.

- The class `TestSortedRegister` will be used to check how `SortedRegister` runs.

- The class of utilities `CorrectReading` will be used for reading values of built-in data types from the standard input. This class is used in the previous one.

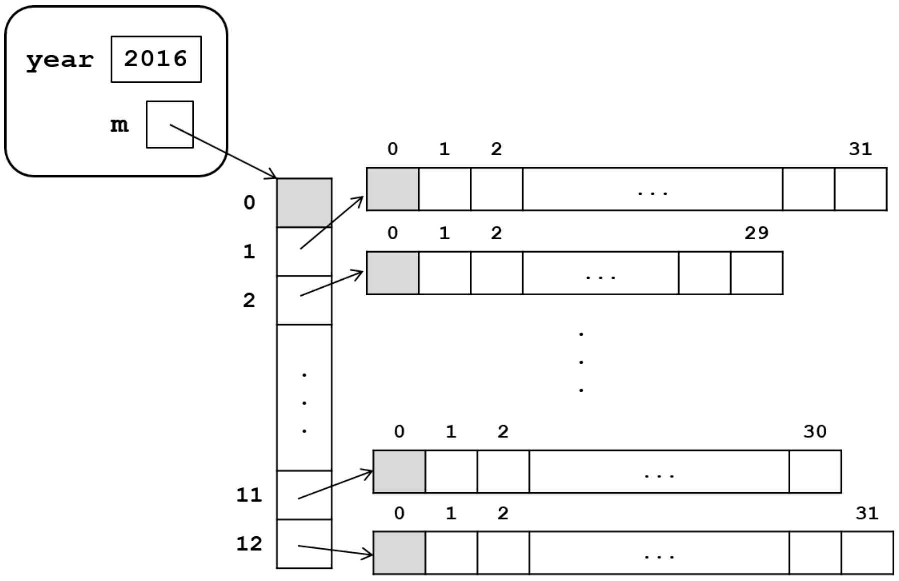

Figure 2: Data structure of objects of the class `SortedRegister`.

3 Checked and unchecked exceptions

Java distinguishes between *checked* and *unchecked* exception types. The *checked* ones must be caught. If a *checked* exception is not caught within one method, it will be ignored explicitly in such method to force it will be caught by one of the methods in the call stack. Catching *unchecked* exceptions is optional, but sometimes it could be helpful, for recovering the program from a failure avoiding stop the execution, or for stopping it adequately.

Checked exceptions appear when it is so difficult to foreseen what was going to happen and it is not possible to avoid the failure. Exceptions related to input/output operations are commonly *checked* exceptions.

This lab practice is organised in two parts and both types of exception are going to be handled:

- Classes `CorrectReading` and `SortedRegister` must be completed in the first part, i.e. the first two sessions. Both types of exceptions can appear in these classes, mainly the ones related to the input operations for built-in data types. Within one method, these exceptions can be ignored explicitly or caught by means of the `try-catch-finally` mechanism.

  Method `main()` from `TestSortedRegister` loads data from the text files provided, the ones that you must use in this lab practice. For simplicity, the exceptions related to input/output operations are not caught, they are explicitly ignored in the profile of `main()`.

- In the second part of this lab practice, you have to face a problem related to the use of the class `SortedRegister`, then, the exceptions related to input/output operations will be handled appropriately.

4 Detection of runtime errors

Activity 1: package `pract4` creation and preparation

- Create a package named `pract4` in the project `prg`.
- Download from the folder `Recursos/Laboratorio/Lab Practice 4` in the `PoliformaT` site of `PRG`, the Java files
  - `CorrectReading.java`
  - `SortedRegister.java`
  - `TestSortedRegister.java`

  then, add them to package `pract4`.
- Download from the same place the files `data2016.txt` and `badData.txt`, then copy them into the folder corresponding to project `prg`.

Activity 2: error detection during runtime for class `CorrectReading`

- Method `nextDoublePositive(Scanner, String)` reads values of type `double` ≥ 0. It uses internally a `do-while` loop that iterates while the read value is < 0. The second argument is an `String` with the prompt to be show to the user when asking for the value.
- Check this method in the (*Code Pad*) of *BlueJ*. To do it, run the following sentences:

  ```
  import java.util.*;
  Scanner t = new Scanner(System.in).useLocale(Locale.US);
  double realPos = CorrectReading.nextDoublePositive(t, "Value: ");
  ```
- Type negative real values from the *terminal window* of *BlueJ*. Notice that the execution does not finish till a positive value or zero is entered.
- Check it again but introducing an incorrect sequences of characters for a double, for instance `23.r4`.

The execution finishes and a message indicating what happened and where in the code is shown. It is a runtime error or *exception*. Next sections are dedicated to handle this type of errors.

5 Catching exceptions. A first example

Several activities are proposed in this section for handling predefined exceptions. You must catch them by using the `try-catch-finally` mechanism. Thanks to this, the following methods allow us to nicely react to certain errors users do frequently when they type numeric values.

Activity 3: analysis of method `nextInt(Scanner, String)`

- Given the method `nextInt(Scanner, String)` from class `CorrectReading` for reading integer values. The read process is done by using method `nextInt()` from class `Scanner`. This method can throw the exception `InputMismatchException` when the character sequence entered by the user is not valid to be converted to an integer.

- Read the documentation of method `nextInt()` of the class `Scanner` in the Java API documentation, and focus your attention in exceptions this method can throw:

  `http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`

- Read also the documentation of the class `InputMismatchException`, verifying it is an *unchecked* exception derived from the class `RuntimeException`, and that the position of both classes matches with the following hierarchy. That is why it is not mandatory to catch them nor explicitly ignore them.

```
java.lang.Object
 |
 +--java.lang.Throwable
     |
     +--java.lang.Exception
         |
         +--java.lang.RuntimeException
             |
             +--java.util.NoSuchElementException
                 |
                 +--java.util.InputMismatchException
```

- Nevertheless, the method `nextInt(Scanner, String)` catches this type of exception by showing an error message to inform the user which action must be done for correcting the error.

  Use the *code zone* of *BlueJ* for running the method "`nextInt(Scanner, String)`" and, from the *terminal window*, type a non-valid sequence of characters to be converted into an integer. A floating point value, for instance. See the message and notice the execution of the method does not finish till a correct integer value is entered.

- Look at the `finally` clause in method `nextInt(Scanner, String)`. Even when an error occurs within a method, they could be pending instructions whose execution is necessary before the method or the program ends. The `finally` clause is just for that, all the instructions in the `finally` block will be executed independently if the `try` block ends with no errors or any exception occurs, again, independently there is a catch block for such exception or not.

  In method `nextInt(Scanner, String)`, the instruction `tec.nextLine()` in the `finally` block is executed in any case. This allows to ignore the new line stored in the input buffer every time the user press the enter key, or the character sequence entered is not valid for the built-in data type. In the last case, an exception of the class `InputMismatchException` is thrown. The instruction `tec.nextLine()` avoids infinite loops.

Activity 4: exception handling in `nextDoublePositive(Scanner, String)`

- Complete the method `nextDoublePositive(Scanner, String)` in the class `CorrectReading` to catch exceptions of the class `InputMismatchException` if the value introduced by the user is not a `double`. Similarly to what it was done for method `nextInt(Scanner, String)`, showing an error message instead of aborting the execution.

With the previous activities you added an exception controller for handling exceptions locally, i.e. within the same method the error/failure was produced.

Activity 5: handling exceptions in `nextInt(Scanner, String, int, int)`

- Complete the method `nextInt(Scanner, String, int, int)` in the class `CorrectReading` for catching exceptions of the class `InputMismatchException` when the value introduced by the user is not an integer. Similarly to what it was done for method `nextInt(Scanner, String)`, showing an error message instead of aborting the execution.

- Moreover, this method must control that the value introduced by the user is in range `[lInferior,lSuperior]`. There two ways for performing this control: The first one is to add the appropriate condition in the loop, as in the case of method `nextDoublePositive(Scanner, String)`. The second one is throw an exception. In this case we will throw an exception, so you have to add a conditional instruction for checking if the introduced value in the range, if it does not belongs to the range, then the method must throw an exception of the class `IllegalArgumentException` by means of the Java instruction `throw` and with a message indicating that the value is not in the range.

  Next, add a `catch` block for locally catching such exception in a way similar to the catching of exception `InputMismatchException`. Showing the message in the exception by using the method `getMessage()` that is available in the class `InputMismatchException` because it is a method inherited from the class `Throwable`.

6 Explicitly ignoring exceptions vs locally handling them

In the class `SortedRegister` described in Section ?? we find different versions of a method where some kind of exceptions are locally handled or explicitly ignored according to the expected behaviour for each one.

Activity 6: checking the behaviour of class `SortedRegister`

Watch the code of the class `SortedRegister.java` downloaded from `PoliformaT`. Its data structure and the methods `add(Scanner)` and `save(PrintWriter)`.

The class `TestSortedRegister` contains the method `testUnreportedSort` for testing the methods of `SortedRegister`. The method `main` reads a correct value for the year in a specified range and the name of a text file containing data, then it creates an object of the class `Scanner` for reading from the text file, and creates an object of the class `PrintWriter` for saving the results. Then `main` invokes the method `testUnreportedSort` for processing the data.

Check to run the class by introducing 2016 for the year, and the file `data2016.txt`. Compare the contents of such file with the one created as a result.

Activity 7: throwing exceptions in the method `add`

In the previous execution, method `add` did not detect the errors contained in the file `data2016.txt` where some negative values for `count` appear. See an example:

```
....
30  4  2
....
30  4  -1
....
```

so the in the text file with the results, i.e. `result.out`, the following line is saved:

```
30  4  1
```

In this case, we expected the method should have rejected the data source and aborted the process.

In order to correct this mistake, you have to modify the method `add` in the class `SortedRegister`, in such a way that when it reads a line from the input stream, if the value read for `count` is negative, it must not be aggregated in the two-dimensional array, and the method must throw an exception of the class `IllegalArgumentException`, derived from `RuntimeException`), with the message `Negative value`.

Execute again the method and check that now the error is detected and the output file `result.out` is empty.

Activity 8: ignoring explicitly exceptions

In addition to the error handled in the previous activity, other errors can be produced when the method `add` is executed. Specially when data do not fulfil the preconditions.

- The method `nextInt()` of the class `Scanner` throws an exception of the class `InputMismatchException`, when it tries to read an integer value in a line like the following one:

  ```
  30  abril  2
  ```

- If `month` and `day` do not correspond to a valid date an exception `ArrayIndexOutOfBoundsException` is thrown when accessing to the two dimensional array, as it can happen in the following example if the year is not a leap year.

  ```
  29  2  1
  ```

The method `add` do nothing in such wrong scenarios, it explicitly ignores such exceptions as explained in the comments associated to this method.

In order to complete the documentation, you have to add the following information:

```
    @throws IllegalArgumentException if a negative value is read from stream s.
```

Notice that these are *unchecked* exceptions, so it is not necessary to explicitly ignore this type of exceptions in the profile of the method.

In order to carry out checks in relation to these types of exceptions, it is available the text file `badData.txt` which contains some errors. Some error examples:

```
...
29  2  1
...
30 abril 2
...
```

Check now to run `TestSortedRegister` in the following cases:

- Year 2016 and text file as data source `badData.txt`. This must produce an exception when the line containing the token `abril` is read. Compare the contents of the file used as input with the file containing the results created by the execution of the class `TestSortedRegister`.
- Repeat the execution with the same file, but introducing 2018 as the year. Then you can see how the execution is aborted as soon as it is processed the line containing the date 29 of February.

In summary, as soon as during the test of `add` it is detected a wrong line in the input text file, any of the previous explained possible errors, then the process is aborted and the instruction for saving the results is never reached. In case of error any data is saved in the results file.

Activity 9: remote exception handling from method `add`

As previously seen, method `add` ignores any exception that can be thrown within its body. `unchecked` exceptions because they are not caught inside the method, `checked` ones because they are explicitly ignored. Due to this, methods `testUnreportedSort` and `main` from `TestSortedRegister` must catch them or explicitly ignore them. And then, the program can be aborted due any of such types of exceptions.

In order to provide more information to the user, you have to modify the method `testUnreportedSort` for catching all the exceptions that method `add` can throw. Depending on the case, one of the following messages must be shown:

```
Wrong file: negative value for the count.
Wrong file: incorrect format for an int.
Wrong file: non-valid date.
```

Check by running the class `TestSortedRegister` that the obtained message is the correct one in the following cases:

- Year 2016, input filename `data2016.txt`.

- Year 2016, input filename `badData.txt`.
- Year 2018, input filename `badData.txt`.

Activity 10: local exception handling

In this activity method `add` is going to be overloaded by implementing a new method `add` that instead of stopping the data loading when a line with errors is read, it completes the process by filtering the lines with errors. An error report is generated.

```
    /** Sort data read from Scanner s.
     *  Wrong data is filtered, an error report is printed.
     *  Precondition:
     *      The accepted and non-filtered data follows this format:
     *        day month count
     *  where day and month must be integers corresponding to a correct date
     *  and count must be > 0.
     *  The count must be aggregated in the register of the day and month.
     *  Wrong lines are reported via err indicating the line number.
     *
     *  @param s Scanner data source
     *  @param err PrintWriter stream where report errors
     */
public void add(Scanner s, PrintWriter err)
```

For it, this method must be a modification of the previous one such that:

- Accounts the number of processed lines from `s`.
- For each line from `s`, and inside a `try` block, this method must try to get the data from the current line and aggregated the read count in the component of the two dimensional array corresponding to the date specified in the line.
- Catch the thrown exceptions, reporting in `err` one of the following messages depending on the error:

  ```
  Line n: negative count
  Line n: incorrect format
  Line n: wrong date
  ```

  where `n` is the line number where the exception was thrown.

For checking that, add a new method in the class `TestSortedRegister` with the profile:

```
public static void testReportedSort(int year, Scanner in, PrintWriter out,
    PrintWriter err)
```

that must create an object of the class `SortedRegister` for a given `year`, then aggregate into this object all the data read from the input stream `in`, writing the results sorted chronologically in the output stream `out`, and reporting the errors found in `err`.

The method `main` from the class `TestSortedRegister`, after reading the year and the filename, opening the input file and creating the output file `result.out`, must ask the user to choose on the following options:

```
Sorting options:
  1.- Reject the file if contain errors.
  2.- Filter the wrong lines.
```

In case 1 it must be used the method `testUnreportedSort`. In case 2 an error file must be created, with name `result.log`, and the method `testReportedSort` must be used, writing the errors found in the file `result.log`. Remind that at the end, the output streams must be closed, i.e. the object of the class `PrintWriter`.

Check to run the option 2 with the following configuration:

- Year 2016, filename `data2016.txt`.
- Year 2018, filename `badData.txt`.

then, check the contents of files `result.out` and `result.log`. In one of the cases `result.out` will be empty, and `result.log` will be empty in the other.

7 Evaluation

This lab practice belongs to the second block of lab practices of PRG, it will evaluated during the second midterm exam. The weight of this block is 60% in relation to the final lab grade. The final lab grade is the 20% of the final grade of PRG.