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

Lab activity 4. Exception management and files
Second part: Obtention of a sorted accident record from a data file

(Two sessions)

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Contents

1 Problem statement

There is available a file with the amount of accidents that happened during a whole year. The record can be given from several areas (towns, provinces, ...), and data can be distributed in one or more text files, where each line has the following format:

```
day  month  amount
```

where `day` and `month` are integers, higher than 0, that correspond to a date in the year, and `amount` is a non-negative integer that corresponds to the amount of accidents recorded in that date.

In a same data file, repeated dates can appear, and lines are not necessarily in chronological order, as it could happen when a file is the result of concatenating the files from different areas.

It is needed an application that extracts data of a given year from one or more text files, and that generates a result file in which appear, in chronological order, the cumulative data for each date with records, in such a way that they appear as Figure 1 shows. It must be taken into account that the files may contain wrong annotations, because they have a line with less or more than three values, or because they are not integer, or because `day` and/or `month` are not a correct date, or because `amount` is negative.

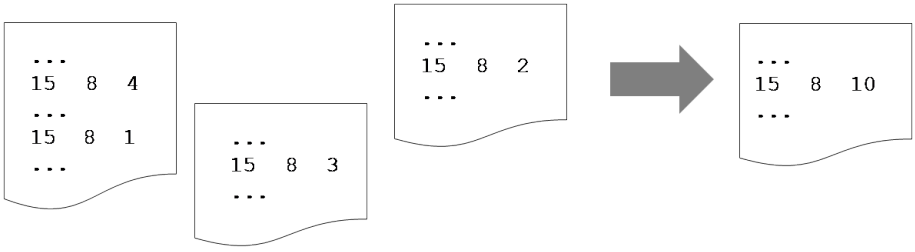


Figure 1: Data aggregation from one or several files.

To solve the problem, it is proposed to employ a matrix whose rows are indexed by months and whose columns are indexed by the day of the month, is such a way that the amount that appears in each line of the processed data is directly accumulated on the component of the matrix indexed by the month and day of the date. After collecting data on the matrix, a listing process on rows and columns would allow to obtain a list, sorted by date, of the accumulated data.

2 Application classes

In the material for the lab activity, you are given the datatype class `SortedRegister`, whose functionality allows to process data with a nature similar to that previously presented.

Objects from the class `SortedRegister` (see Figure 2) have a bidimensional array `m` in which rows are the months and columns are the days of each month, in such a way that `m[r][c]` is used to store the accumulated accidents for day `c` of month `r`. Rows 1 to 12 correspond to the months of the year (row 0 is not used). For each row, columns from 1 on correspond to the days of the month (column 0 is not used). Notice that the last column for February should be 29 or 28 depending on it is a leap year or not.

The most important methods implemented in this class are:

- Constructor

  ```
  public SortedRegister(int year)
  ```

  which creates the matrix `this.m` according to the given year (`year`).

- Method with header

  ```
  public int add(Scanner sc)
  ```

which, given a `Scanner` object `sc` that was opened from the text data source, processes each of the lines from `sc` to cumulate data on `this.m`. When the process finishes normally (without exceptions raised because of wrong data format), it returns the number of processed lines.

- Method with header

```
public void save(PrintWriter pw)
```

that writes into `pw` the accumulated data present in `this.m`, in chronological order.

To test the behaviour of `SortedRegister`, by reading data from specific data files, you have available the program class `TestSortedRegister`. This class employs the utility class `CorrectReading`, developed during the first part of this lab activity.

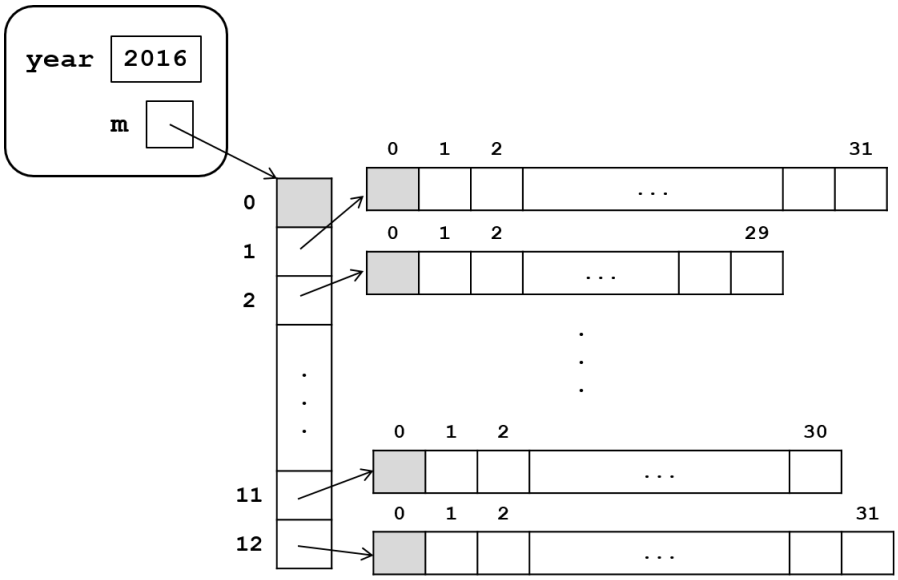


Figure 2: `SortedRegister` structure.

2.1 Activity 1: installing classes and test files

- Download from the lab activity 4 folder the files `SortedRegister.java` and `TestSortedRegister.java`; add them to the `pract4` package.
- From the file browser and in the `prg/pract4` folder, create a new folder with name `data`, where data files must be moved and result files would be generated.
- Download from the same *PoliformaT* folder the text files that would be used for testing: `data.txt`, `badData1.txt`, `badData2.txt`, `badData3.txt`, `badData4.txt`. Copy them into folder `prg/pract4/data`.

It is important to situate data files in the correct place: as next activity will show, the program `TestSortedRegister` searches the files for the tests in the folder `pract4/data` of the `prg` project. Result files are saved in the same `data` folder.

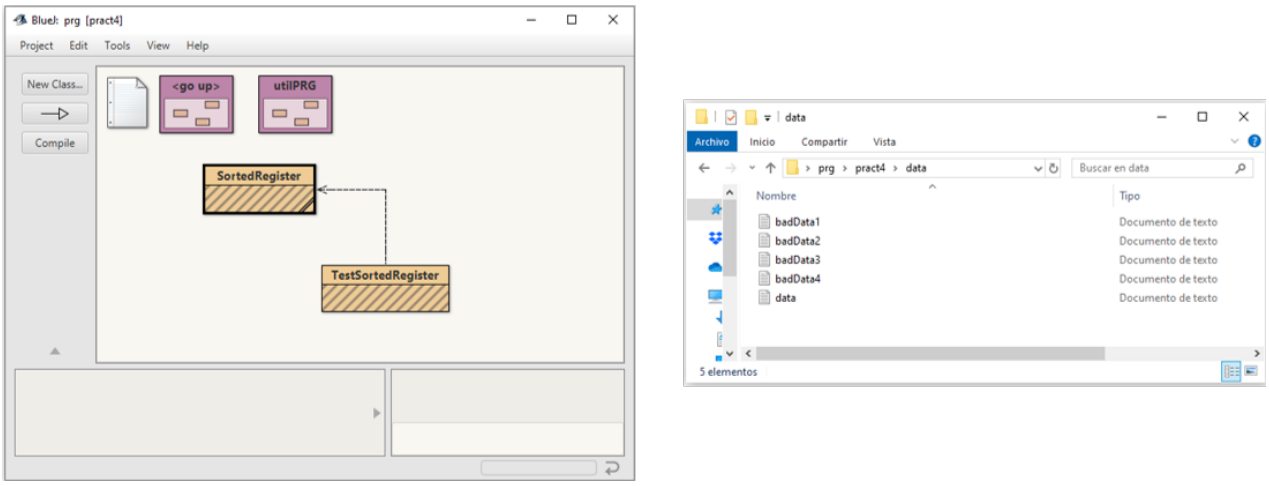


Figure 3: Package `pract4` with the application classes and the `data` folder with data files.

2.2 Activity 2: inspection and test of the `SortedRegister` class

Inspect the code for the `SortedRegister` class downloaded from *PoliformaT*: data matrix structure, `add` method (which employs an auxiliar method `handleLine` to obtain data from each line and to update with them the `this.m` matrix), and `save` method.

Inspect the `TestSortedRegister` class, which contains a method `main` and a method `test1` that is used to check the methods from `SortedRegister`. The `main` method is used to read a correct year in a given interval and the name of a data file, to open a `Scanner` and a `Printwriter` object from the data and result (`result.out`) file, respectively, and it uses the `test1` method to process data.

Check the class execution, using as data year 2016 (leap year), the `data.txt` file, and selection option 1 from the menu (`test1`). Compare the content of the input file with that of the result file. As shown in Figure 4, you can observe that in `result.txt` appear the same data than in `data.txt`, but correctly accumulated and in chronological order.

Warning: when executing the `main` method of `TestSortedRegister` input/output exceptions may appear when the name of the file is incorrectly written or it is not in the expected folder. These exceptions would be handled in the future, as Section 4 explains.

3 Handling `RunTimeException` exceptions

3.1 Activity 3: inspection and test of the `handleLine(String)` method

For processing each of the lines that the `add` method reads by using `nextLine()`, it calls the private auxiliar method `handleLine(String)`, where the line is the parameter.

When inspecting the code of this auxiliar method, it can be seen that the first action is applying the method `split`, from the `String` class, to `line`. This is done to "split" the line in substrings separated by blank spaces: `split` returns an array whose components are the different substrings that appear in the line, in a way such that if `line` fits the format, the resulting array has exactly three components.

If the three integers `day`, `month`, and `amount` have been correctly extracted, the read amount is accumulated on the matrix component `this.m[month][day]`.

Notice that this method throws the exceptions that correspond to the following errors:

data.txt

```
 2   1   1
29   2   1
23   3   1
30   4   2
12   5   2
25   6   2
15   8   3
30   8   1
23   9   2
30   9   1
23  10   6
 1  11   3
 4  12   1
31  12   3
 2   1   1
31   1   2
28   2   3
24   3   1
30   4  -1
12   5   2
15   6   4
30   6   1
23   7   6
31   7   1
15   8   7
31   8   3
30   9   1
23  10   6
 4  12   1
31  12   3
```

result.out

```
 2    1    2
31    1    2
28    2    3
29    2    1
23    3    1
24    3    1
30    4    1
12    5    4
15    6    4
25    6    2
30    6    1
23    7    6
31    7    1
15    8   10
30    8    1
31    8    3
23    9    2
30    9    2
23   10   12
 1   11    3
 4   12    2
31   12    6
```

Figure 4: Contents of the text files `data.txt` and `result.out`.

1. When the splited line (the result of applying the `split` method) has not exactly three data items, it creates and raises an exception from the class `IllegalArgumentException`, with the message: `"Line does not contain three data items."`

2. When any of the three data items from the line cannot be transformed into `int`, it throws the `NumberFormatException` exception caused by the `Integer.parseInt` method.

3. When `day` and/or `month` are not indexes ≥ 1 of a component of the matrix `this.m`, it creates and raises an `IllegalArgumentException` exception with the message `"Incorrect date"`. Remember that the `SortedRegister` constructor creates the `this.m` matrix with rows 1 to 12 corresponding to the months of the year, and for each of these rows, it creates columns from 1 on corresponding to the days of the month.

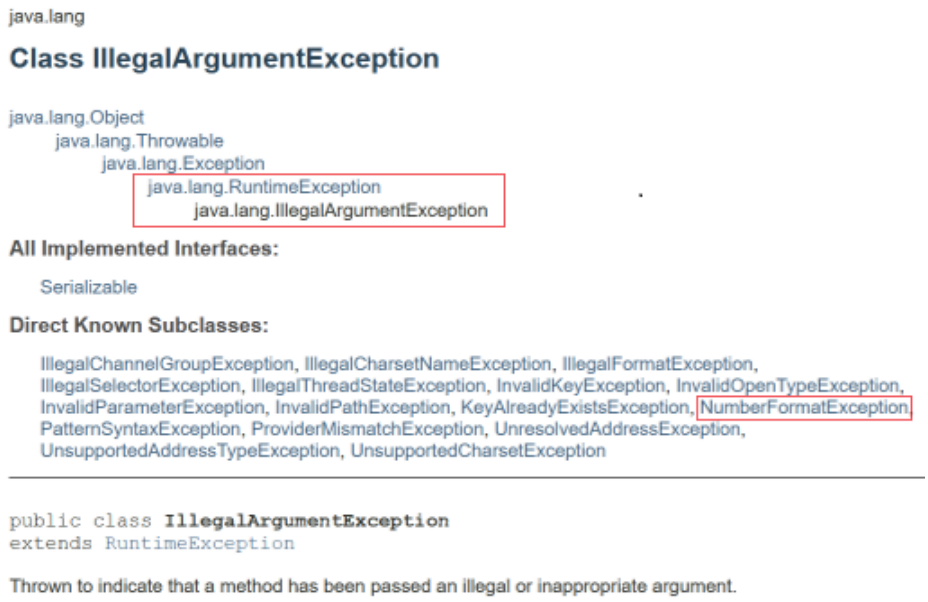
For the previous 1 and 3 situations, it is chosen to create an unchecked exception from the class `IllegalArgumentException`. Its use, as documentation in Figure 5 shows, is appropriate when a method receives a wrong parameter. The created message allows to distinguish between the two possible situations.

To check exception throw to `add`, and from that point to the program, make the following test:

Execute again `main` for the `TestSortedRegister` class, option 1, but passing as data year 2016 and file `badData1.txt`. Look at the exception raised (thrown by `handleLine`).

Then, inspect the contents of `badData1.txt`, and check that the first wrong line that contains the file corresponds with the raised exception.

Check that `result.out` is empty, since `test1` gets interrupted without arriving to the instruction that writes the result file.



java.lang

Class IllegalArgumentException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IllegalArgumentException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

IllegalChannelGroupException, IllegalCharsetNameException, IllegalFormatException, IllegalSelectorException, IllegalThreadStateException, InvalidKeyException, InvalidOpenTypeException, InvalidParameterException, InvalidPathException, KeyAlreadyExistsException, NumberFormatException, PatternSyntaxException, ProviderMismatchException, UnresolvedAddressException, UnsupportedAddressTypeException, UnsupportedCharsetException

---

```
public class IllegalArgumentException
extends RuntimeException
```

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Figure 5: Part of the documentation for `IllegalArgumentException`.

3.2 Activity 4: finishing the `handleLine(String)` method

The method `handleLine` that you inspected and tested in the previous activity is not complete. Actually, review the test execution of the previous activity 3, whose result is shown in Figure 4. It can be noticed that in `data.txt` appear the following lines for April 30th:

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

The process of the two lines produced in `result.out` a line that is the result of adding the amounts `2` and `-1`:

```
30  4  1
```

That is, the `handleLine` method did not detect the wrong amount (the amount of accidents in a given date cannot be lower than 0).

Thus, in order to make `handleLine` to manage all the possible errors, a statement must be added to the method such that before accumulating in the matrix the read amount, it must check whether it is negative; in that case, it must create and raise an `IllegalArgumentException` exception with the message `"Negative amount."`.

Execute again the `main` method from the `TestSortedRegister` class, option 1, passing as data the year 2016 and the file `data.txt`. Check that just when the line with negative amount is processed the corresponding exception is raised (Figure 6).

3.3 Activity 5: catching exceptions in the `add(Scanner)` method

The method `add` implemented in the class has as a precondition that lines read from the `Scanner` argument fit the defined format. As it has been seen in the previous activities, when a wrong line is read, the method just throws the exception thrown by `handleLine`. In the test program,

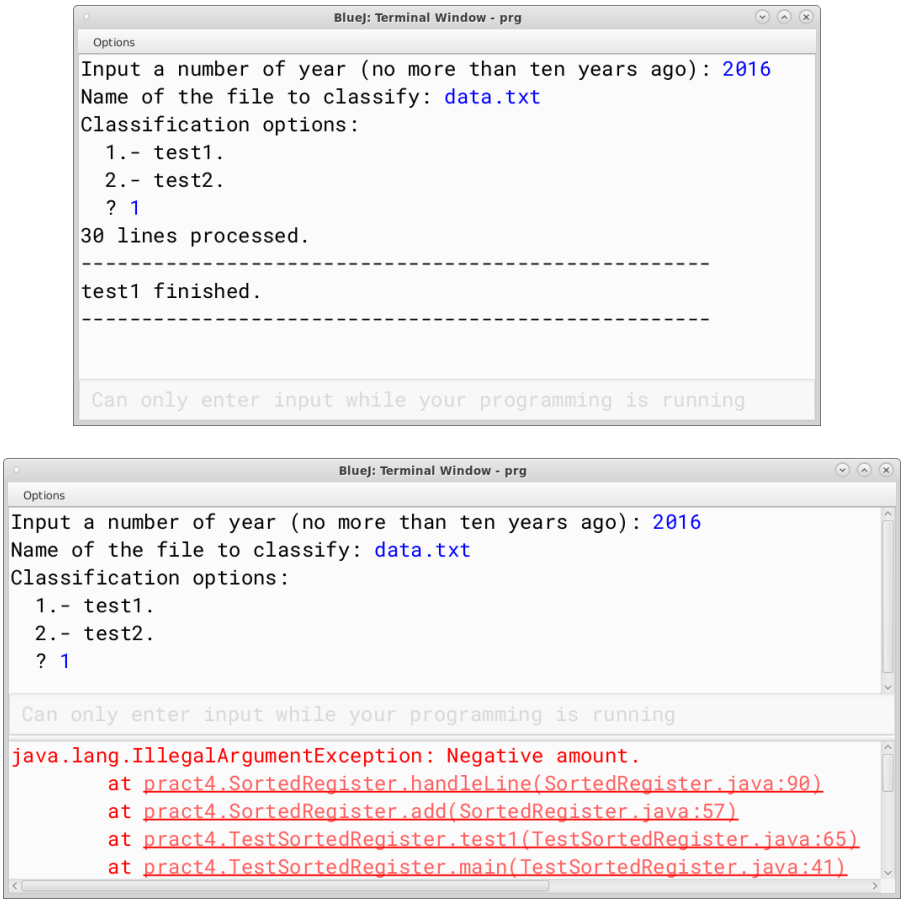


Figure 6: Raising and throwing an exception caused by amount < 0: top execution is previous to `handleLine` completion (activity 4), and bottom execution is posterior to completion.

`TestSortedRegister`, this causes that methods `test1` and `main` throw it themselves, which causes execution termination.

To improve the behaviour of the method to obtain a normal program end (without exception throws) it must be done in a way such that:

- When all lines are correct, the method returns the amount of processed lines.
- Otherwise, just when it detects from the `Scanner` source any line with a wrong format, it will interrupt the process and it will return -1. Before terminating, it must write on standard output one of the following messages, according the detected error:

```
ERROR. Line n: Line does not contain three data items.
ERROR. Line n: No integer data item.
ERROR. Line n: Incorrect date.
ERROR. Line n: Negative amount.
```

  where `n` is the number of line where the error was detected.

In order to do that, the method body must follow a structure such like:

```
int count = 0;
try {
```

```
        ...
    } catch (NumberFormatException e) {
        System.out.println("ERROR. Line " + count + ": No integer data item.");
        count = -1;
    } catch (IllegalArgumentException e) {
        System.out.println("ERROR. Line " + count + ": " + e.getMessage());
        count = -1;
    }
    return count;
```

It is important to have into accout that the `NumberFormatException` is an exception derived from `IllegalArgumentException` (see Figure 5); thus, the compiler demands for a `catch` instruction of this exception **previous** to the `catch` of its base class.

Method documentation and comments should reflect these changes (i.e., that the method stops data read if any wrong line is detected and that returns -1 in that case, and erase those lines labeled with `@throws`).

With this modification, when the execution of `test1` is repeated with the file `data.txt`, it finishes normally, as it happens in the example shown in Figure 7. It can be seen that in the code for `test1`, if `c.add(sc)` returns a negative value, then data in `c` is not saved into `PrintwWriter out` (it remains empty).

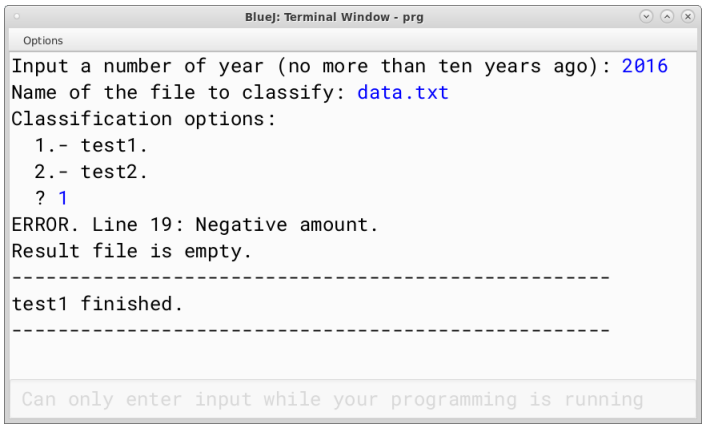


Figure 7: Exception catching for `add(Scanner)`: when a wrong line appears, `add` does not throw the corresponding exception.

To test the correct behaviour of the method, you have available different files (`badData1.txt`, `badData2.txt`, `badData3.txt`, and `badData4.txt`). The four files contain the same data, with the same four incorrect lines (lines number 4, 5, 6, and 7), but in different order. In that way, the first wrong line in each case corresponds with the four different error cases, respectively. The option 1 of `main` from `TestSortRegister` must be tested, and the corresponding error message that is written:

- Input for year 2016 and file `badData1.txt`.
- Input for year 2016 and file `badData2.txt`.
- Input for year 2016 and file `badData3.txt`.
- Input for year 2016 and file `badData4.txt`.

The file `data.txt` can be edited to erase the incorrect line with a negative amount and obtain a file without errors. If the test is repeated, the method must process correctly the file with the remaining lines.

3.4 Activity 6: development of the `add(Scanner, PrintWriter)` method

The method `add` from `SortedRegister` is overloaded by the method with header

```
public int add(Scanner sc, PrintWriter err)
```

expecting its code to be completed. As before, `sc` is the data source, but now all its lines will be processed filtering the errors, and those lines would be reported into `err`.

Concretely, this method must be a modification of the previous one, in such a way that, for each of the text lines from `sc`:

- It would try to obtain all data from the line and accumulate the read amount on the matrix, by using the `handleLine` method.
- It catches all the exceptions thown by `handleLine`, writing in `err` one of the following messages according the case:

  ```
  ERROR. Line n: Line does not contain three data items.
  ERROR. Line n: No integer data item.
  ERROR. Line n: Incorrect date.
  ERROR. Line n: Negative amount.
  ```

  where `n` is the line number where the exception raises.

In this way, `this.m` would store data of the lines that fit the format, whereas `err` stores the error messages. The method would finish by returning the total amount of processed lines, both correctly or incorrectly.

To test it, complete the `test2` method from the `TestSortedRegister` class. The method must be similar to `test1`, except that it must use the method with header `add(Scanner, PrintWriter)` completed in this activity, and in all cases the data stored in the matrix `this.m` must be saved into `out`. After completing `test2`, execute tests such as those shown in Figure 8.

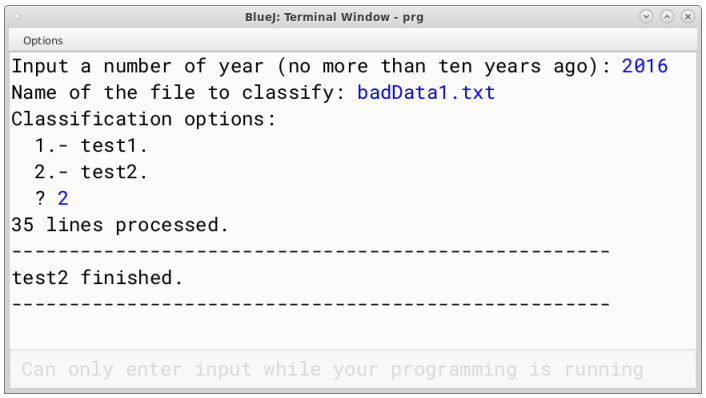


Figure 8: Testing the execution for `add(Scanner, PrintWriter)`.

For this execution, the resulting files `result.out` and `result.log` should be those presented in Figure 9.

badData1.txt

```
 2   1   1
 1   6   1
23   3   1
30   1   4   5
30  abril   2
31   9   2
30   4  -1
12   5   2
25   6   2
15   8   3
30   8   1
23   9   2
30   9   1
23  10   6
 1  11   3
 4  12   1
31  12   3
 2   1   1
31   1   2
28   2   3
24   3   1
29   2   1
12   5   2
15   6   4
30   6   1
23   7   6
31   7   1
15   8   7
31   8   3
30   9   1
23  10   6
 4  12   1
31  12   3
16   5   1
 2   1   2
```

result.out

```
 2   1   4
31   1   2
28   2   3
29   2   1
23   3   1
24   3   1
12   5   4
16   5   1
 1   6   1
15   6   4
25   6   2
30   6   1
23   7   6
31   7   1
15   8  10
30   8   1
31   8   3
23   9   2
30   9   2
23  10  12
 1  11   3
 4  12   2
31  12   6
```

result.log

```
ERROR. Line 4: Line does not contain three data items.
ERROR. Line 5: No integer data item.
ERROR. Line 6: Incorrect date.
ERROR. Line 7: Negative amount.
```

Figure 9: Result for the test of Figure 8.

4 Handling `IOException` exceptions

Java distinguishes between *checked* exceptions (that must be handled by catching or throwing them) and *unchecked* exceptions (that do not require handling, which are derived from `RuntimeException`). Checked exceptions appear in situations where the error cannot be foreseen and avoided, which is typical when accessing files.

In the `main` method of the `TestSortedRegister` program, the `FileNotFoundException` checked exception may appear when there is any problem when opening `Scanner` or `PrintWriter` objects employed in the tests (no permision, wrong file name, wrong path, not enough memory, etc.).

In this section it is proposed to catch this kind of exceptions to avoid program termination.

4.1 Activity 7: catching `FileNotFoundException` exceptions in the `TestSortedRegister` program

Inspect the code for the `main` method of `TestSortedRegister`. This method does not present exception catching for the `FileNotFoundException`, which makes the compiler to ask for modifying the `main` method header by adding to it the modifier `throws FileNotFoundException`.

In this activity, you must change the code of the `main` method to avoid throwing the exception. For that, after reading from keyboard year and data file name, you must add the following exception handling instructions:

```
...
Scanner in = null; PrintWriter out = null, err = null;
File f = new File("pract4/data/" + nameIn);
try {
    in = new Scanner(f);
    f = new File("pract4/data/" + "result.out");
    out = new PrintWriter(f);
    f = new File("pract4/data/" + "result.log");
    err = new PrintWriter(f);
    ...
    // Selection and execution of the test
    ...
} catch {FileNotFoundException e) {
    System.out.println("Error when opening file " + f);
} finally {
    if (in != null) { in.close(); }
    if (out != null) { out.close(); }
    if (err != null) { err.close(); }
}
```

It is important to emphasize that the virtual machine always executes the code in the `finally` block. In that way, we get sure that in all cases the files that where opened in the `try` block get closed.

After adding the exception handling in the method, the `throws` modifier can be removed from the `main` header.

Figure 10 shows the behaviour of `TestSortedRegister` without handling the `FileNotFoundException` exception, and after including the handling code.

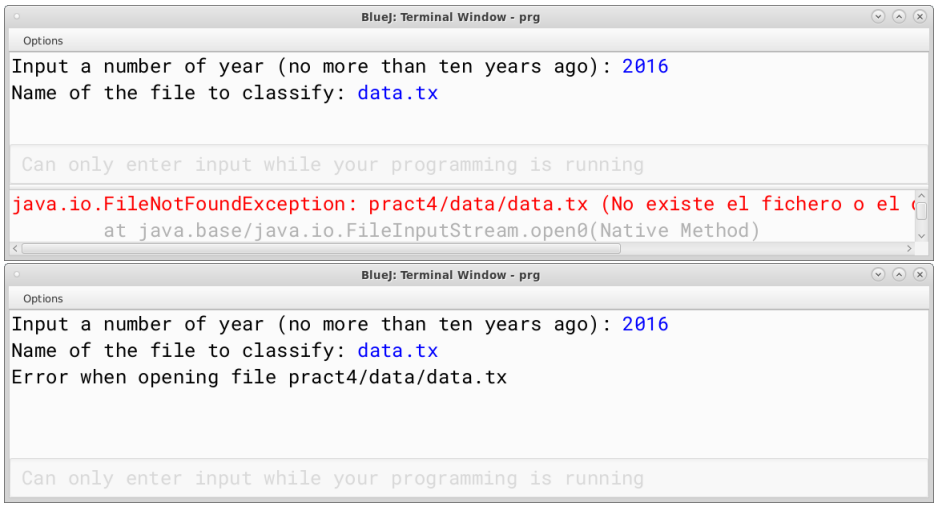


Figure 10: `FileNotFoundException` handling: top execution is previous to `main` (of `TestSortedRegister`) completion according activity 7, and bottom execution is posterior to the completion.