Unit 2. Final exercise

Exam reminder

Service and Process Programming

Arturo Bernal
Nacho Iborra

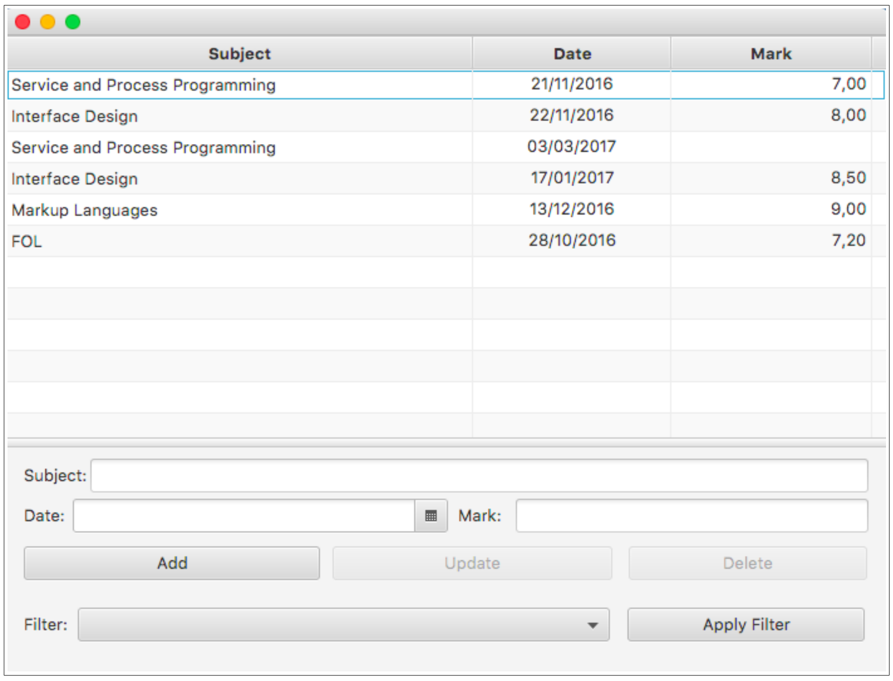
IES San Vicente

Index of contents

1. Introduction and first steps

In this final exercise we are going to implement a JavaFX application to store the exams that we do, so that we can filter them and show some statistics.

The appearance of the application will be more or less like this:



Subject	Date	Mark
Service and Process Programming	21/11/2016	7,00
Interface Design	22/11/2016	8,00
Service and Process Programming	03/03/2017	
Interface Design	17/01/2017	8,50
Markup Languages	13/12/2016	9,00
FOL	28/10/2016	7,20

1.1. Setting up the project

We are going to create a Single Gradle Project called **ExamReminder** to develop this application. Follow the steps explained in Week #07 to create this type of project.

Inside the *Source Packages* section, create the following packages and subpackages:

- *examreminder* package, which will be our main package with the JavaFX main class and controllers
- *examreminder.model* package, to store our model (*Exam* class, as explained later)
- *examreminder.utils* package, to store some useful classes

You can also set the application's main class in *build.gradle* file to *ExamReminder* (we will create this main class later as well).

```
if (!hasProperty('mainClass')) {
    ext.mainClass = 'examreminder.ExamReminder'
}
```

2. Class structure

Besides the JavaFX main application with the FXML files and controllers (we will see them in next section), we are going to need some additional classes to store the information about the exams, and help us do some basic operations.

2.1. The Exam class

Add a new class inside the *examreminder.model* package called **Exam**. This class will have the following attributes:

- The subject of the exam (a *String* with the subject's name)
- The exam date (*LocalDate*)
- The exam mark (*float*)

Besides, we will add two constructors:

- A constructor with the subject name and date (we will set the mark to an appropriate default value)
- A constructor with all the attributes

Add the getters and setters for each attribute.

2.2. FileUtils class

In order to get and save the information from / to text files, we are going to create a class called **FileUtils** in the *examreminder.utils* package. This class will have a set of useful static methods, such as:

- `static List<Exam> loadExams()` to load exams from a given text file
- `static void saveExams(List<Exam> exams)` to save the list of exams in the file

2.2.1. Exams file structure

The structure of the text file containing the exams information will be as follows:

```
subject:date:mark
```

For instance:

```
Service and Process Programming:21/11/2015:8

Data Access:23/11/2016

Interface Design:03/03/2016:7.5

...
```

where the attributes are separated by ':'. If the exam has not been qualified yet, then the mark (and the last ':') will not be present (as you can see in line number 2 of previous example)

2.3. Other useful classes

Although it is not compulsory, it may be useful to add some other classes. For instance, a class called *MessageUtils* (in the *examreminder.utils* package) to show different *Alert* messages. It could have these static methods:

- `static void showError(String message)` to show error messages
- `static void showMessage(String message)` to show information messages
- ...

You can add as many classes as you need inside this package.

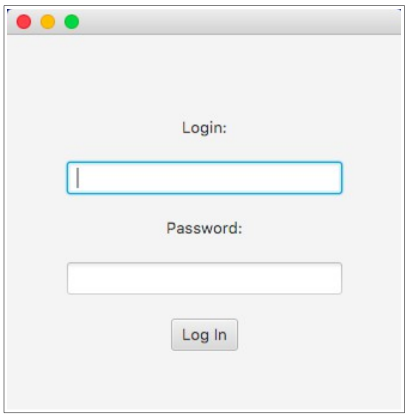
3. The JavaFX application

Let's create the JavaFX application classes. Follow these steps:

1. Create a new JavaFX Main Application called **ExamReminder** inside *examreminder* package (remember that, in section 1, you set up Gradle so that your main class should be called *ExamReminder*).
2. Create a new FXML file called **FXMLMainView.fxml** with its associated controller (**FXMLMainViewController.java**). The FXML file will be placed inside the *resources/fxml* folder by default, and the controller can be placed in the *examreminder* package, as you did with the main application.
3. Create another FXML file called **FXMLLoginView.fxml** with its associated controller (**FXMLLoginViewController.java**). The FXML and controller will be placed in the same folders than the ones for the main view (step 2).
4. Change the code of the *start* method of *ExamReminder* class to make it load the contents from the FXMLLoginView file, as explained in the tutorial of Week #07.

3.1. Designing the login view

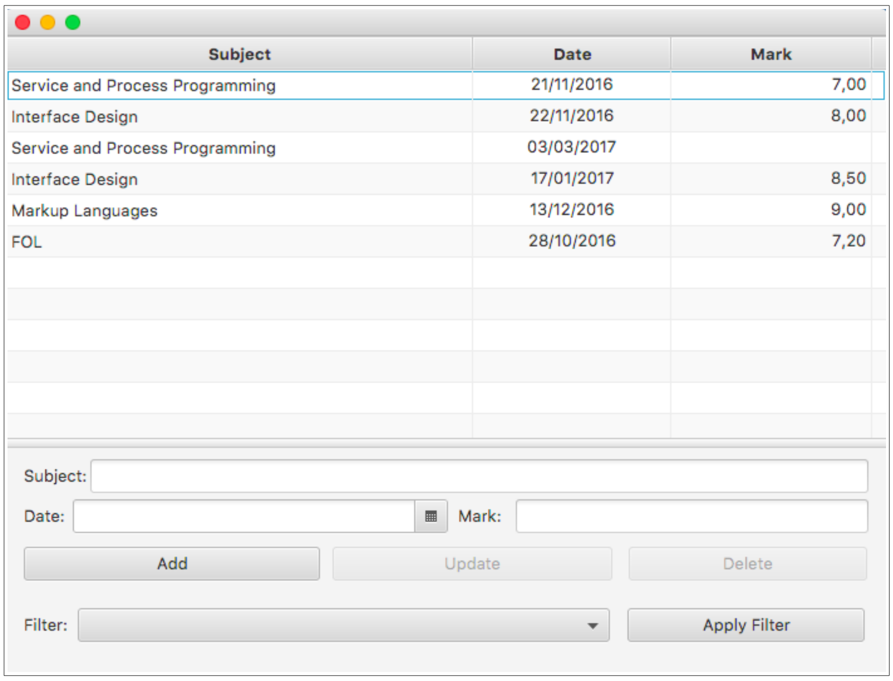
Use Scene Builder to design the login view. Use a *VBox* layout as the main container for this view, and place some labels, text fields and button to get something like this (you may need to add some alignment, padding and spacing in the *VBox* to get the controls with this appearance):



You can use a *PasswordField* control for the *Password* text field, instead of using a simple *TextField*.

3.2. Designing the main view

Use now Scene Builder to design the main scene and get an appearance similar to this:



Subject	Date	Mark
Service and Process Programming	21/11/2016	7,00
Interface Design	22/11/2016	8,00
Service and Process Programming	03/03/2017	
Interface Design	17/01/2017	8,50
Markup Languages	13/12/2016	9,00
FOL	28/10/2016	7,20

You can, for instance, use a vertical *SplitPane* as the main container, and then:

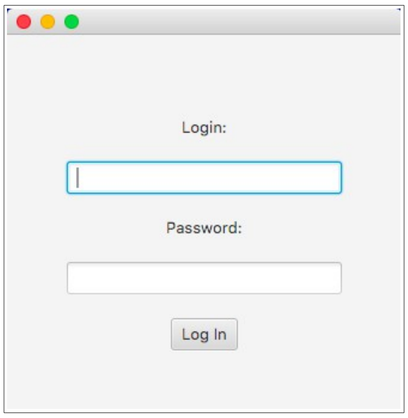
- Place a *TableView* with its corresponding *TableColumns* in the upper section
- Place the form controls below the table view in the lower section (use the appropriate(s) container(s) to display these controls).

Remember to set an *fx:id* to each element that may need to be accessed from the controller.

3.3. How should it work?

3.3.1. Application log in

When we run the application, the login view must be shown:

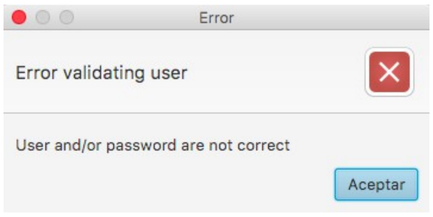


There will be a login and password stored in a text file, with the format:

```
login:password
```

In the file given to you, the user is **admin**, and the password is also **admin**. This password is encoded using SHA256. Read section 4.1 to find out how to encode the password written by the user to compare with the one stored in the file.

When the user clicks on the *Log In* button, the application will load the data from this file, and compare it with the one typed in the text fields. If everything is OK, then we will switch the view to *FXMLMainView* (see Annex 2 to learn how to deal with multiple views). Otherwise, an error message will appear, and the user will have to try again.



Error

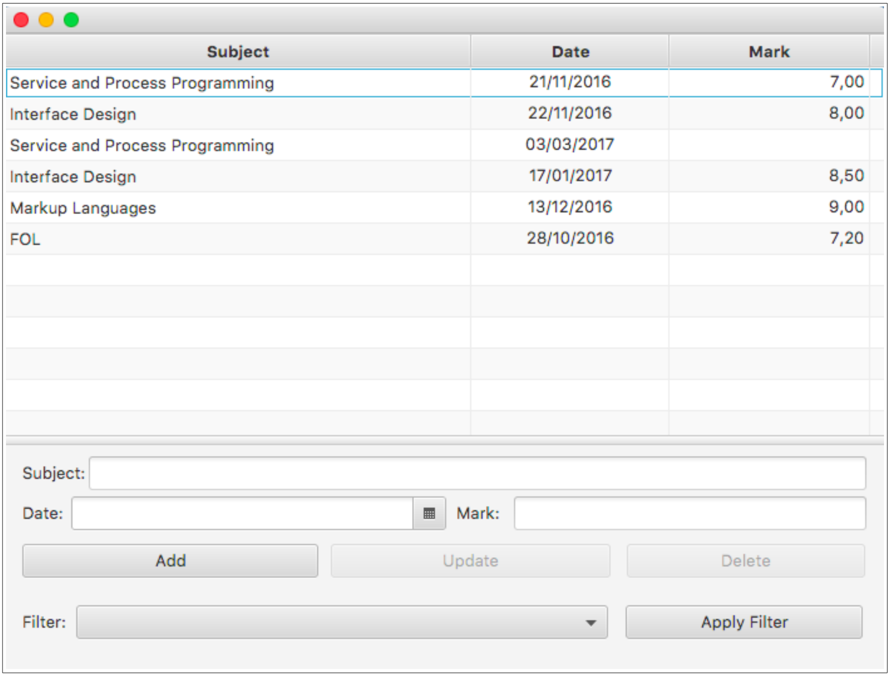
Error validating user

User and/or password are not correct

NOTE: it may be useful to add a method to our *FileUtils* class to read the user and password from the text file and compare them with the ones typed in the text fields

3.3.2. The main View. Adding, updating and deleting exams

Once we validate, we will see the main view:



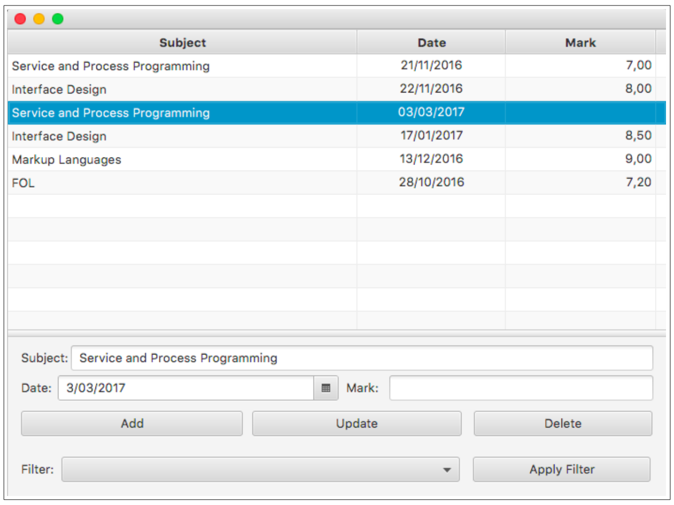
Subject	Date	Mark
Service and Process Programming	21/11/2016	7,00
Interface Design	22/11/2016	8,00
Service and Process Programming	03/03/2017	
Interface Design	17/01/2017	8,50
Markup Languages	13/12/2016	9,00
FOL	28/10/2016	7,20

The application will load the exams from the text file into a list, and the table will show all the exams from that list. Note that, if the exam has no mark, the *Mark* column must be empty.

Then, we can do the following operations:

- If we fill the *Subject*, *Date* and *Mark* fields in the bottom form (*Mark* can be empty), and click on the *Add* button, a new exam will be added to the exam list (and to the table)

- *Update* and *Delete* buttons must be disabled at the beginning (use the *setDisable* method to do this). When we click on any exam from the table, then its data will be copied in the bottom form, and these buttons will be enabled.



Subject	Date	Mark
Service and Process Programming	21/11/2016	7,00
Interface Design	22/11/2016	8,00
Service and Process Programming	03/03/2017	
Interface Design	17/01/2017	8,50
Markup Languages	13/12/2016	9,00
FOL	28/10/2016	7,20

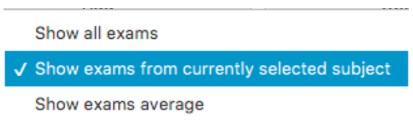
Subject: Service and Process Programming

Date: 3/03/2017 Mark:

  - If we click on the *Update* button, currently selected exam will be update with the data typed in the bottom controls
  - If we click on the *Delete* button, currently selected exam will be removed from the exam list (and table)

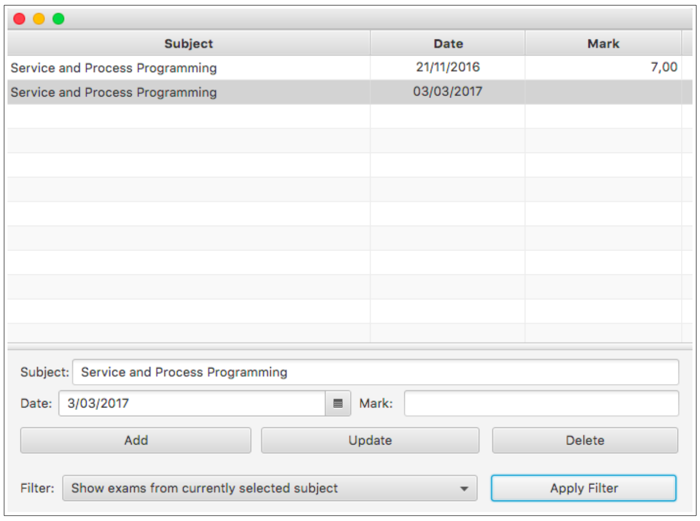
3.3.3. Applying filters

Below the exam form, there is a choice box (or combo box, if you prefer), with a list of filters...



... and an *Apply Filter* button. Each filter will do the following:

- **Show all exams**: it will show every exam in the table
- **Show exams from currently selected subject**: it will show in the table all the exams from the subject typed in *Subject* text field. For instance, if we choose an exam of *Service and Process Programming* (or type this subject in the text field), and we apply this filter, we could get something like this in the table:



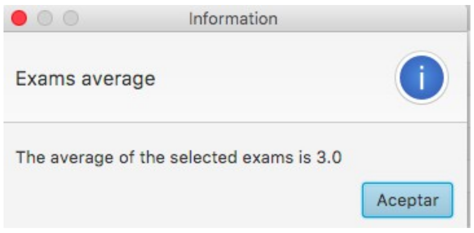
Subject	Date	Mark
Service and Process Programming	21/11/2016	7,00
Service and Process Programming	03/03/2017	

Subject: Service and Process Programming

Date: 3/03/2017 Mark:

Filter: Show exams from currently selected subject

- **Show exams average**: it will show an information message with the average of all the exams currently visible in the table (exams with no mark must be discarded from this operation). Use a Java 8 *stream* to get this average.



Information

Exams average

The average of the selected exams is 3.0

3.3.4. Saving changes

Whenever we try to close the application, we must capture this event and save in the exams file the current exam list (not the one shown in the table, since it may be filtered).

4. Some Gradle issues

To finish, we are going to add some Gradle features to this project.

4.1. Dependencies

We are going to add a *compile* dependency with Apache Commons Codec library (it is available in the Maven repository, so you will only need to add a *compile* line in the *dependencies* section of *build.gradle*).

We are going to use this library to store the user password in the text file. We are going to crypt this password with an SHA 256 key, and use the *DigestUtils.sha256Hex* method of this library to generate this cyphered key. *DigestUtils* is located in the `org.apache.commons.codec.digest` package.

4.2. Tasks

Create a task (type:Zip) called **backup**. This task will store all the txt files in the main directory of the project into a folder called backup (create it manually first). The name of the file will be 'backup-**yyyyMMddHHmmss**.zip' (backup and the current date and time). To generate a date, create a function like this in your build.gradle file:

```
def getDate() {
    def date = new Date()
    def formattedDate = date.format('yyyyMMddHHmmss')
    return formattedDate
}
```

Then, in the task, just call it and append it to the name:

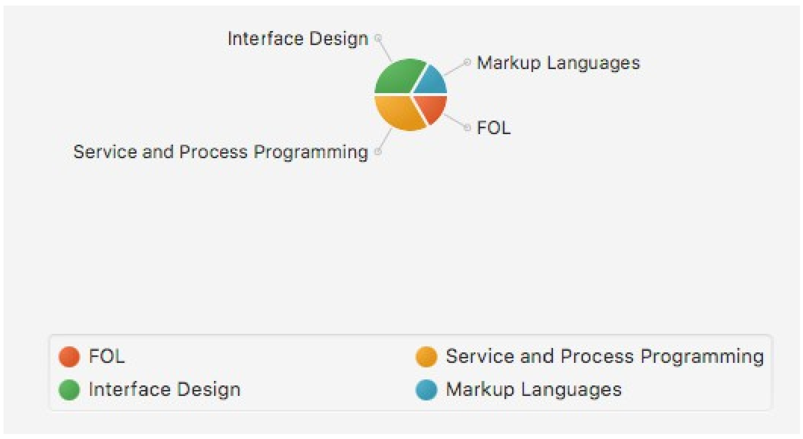
```
'backup-' + datDate() + '.zip'
```

5. Optional improvements

Besides the compulsory part of this exercise, you can try to implement these optional improvements to get a better mark.

5.1. Adding charts

In the left section of *Scene Builder* there is a subsection called *Charts*, with a list of charts that can be added to JavaFX application. Try to add an option to show a *pie chart* with the % of exams of each subject.



You can show this chart in the same main view (increasing the window size), or in another window/dialog.

5.2. Adding CSS styles

Add a CSS stylesheet with some default styles for the main view, such as:

- Background color: choose among any color other than gray, black or white
- Font color: according to the background chosen, it can be either white or gray
- Every button must have a black background with white text color
- Other styles that you may want to add

5.3. Additional filters

You can also add these filters to the filter list at the bottom of the application:

- *Show future exams*: it will show in the table only the exams that have no mark
- *Show maximum marks*: it will show in the table the exam of each subject with the maximum mark

6. Evaluation rules

6.1. Compulsory part

To get your final mark, the following rules will be applied:

- Class structure (*model* and *utils* packages), with the *Exam* and *FileUtils* classes with the corresponding code, and every other additional useful class that you may need: 1 point.
- JavaFX application layout, similar to the one shown in previous figures: 1 point. You must use the appropriate layout containers to arrange the controls properly in the view (HBox and/or VBox are very helpful).
- Application log in and switching from login to main view: 1 point
- Gradle project configuration, including main class set up, dependencies and tasks: 1 point
- Loading exams into the list and table, and adding, updating and removing exams from the list and table (disabling *update* and *delete* buttons when these operations can't be done): 2 points
- Applying filters from the bottom list: 1 point each filter
- Code documentation (Javadoc comments for every class and public method or constructor), cleanliness and efficiency: 1 points

6.2. About the optional improvements

If you decide to implement any optional improvement for this exercise, keep in mind that:

- If your compulsory part is lower than 10, you can get up to 10 points by implementing one (or many) of the optional improvements (up to 1 point per improvement)
- If your compulsory part is perfect (10 points), you can get up to 11 points by implementing ALL the optional improvements.