Object-Oriented Programming 2 – Graded Exercises, Series 1 (DOJ)

Exercise 1 – GUI

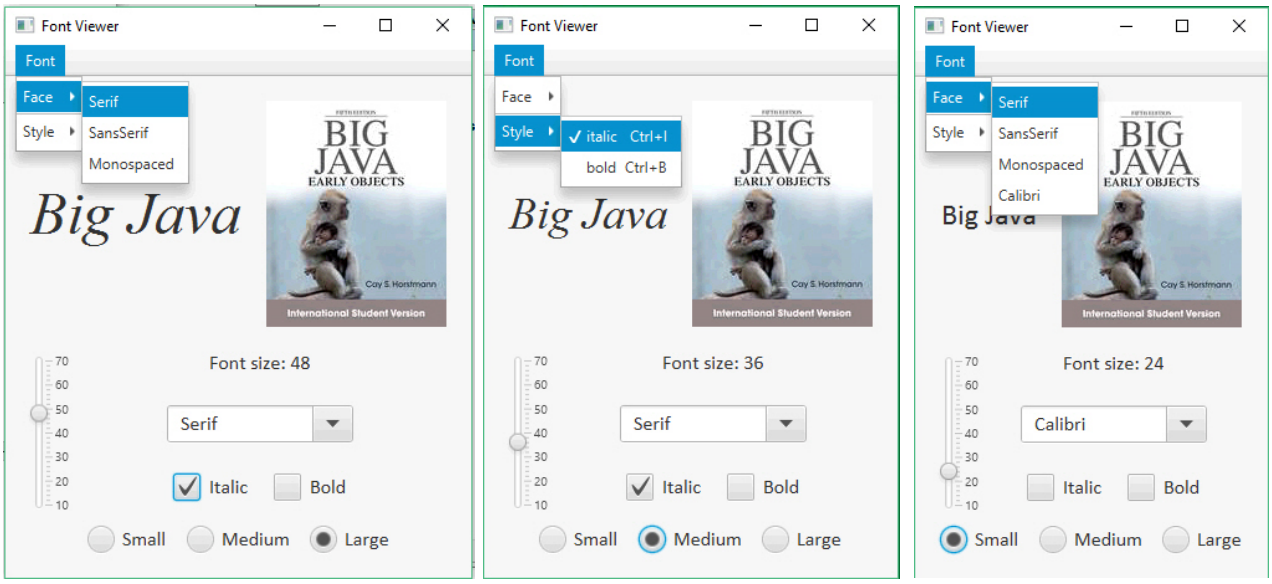
Implement with JavaFX the GUI below. It has the following functionalities:

- The "Big Java" text must always be written according to the selected font type (Serif, SansSerif or Monospaced), style (normal, italic or bold) and size.
  - The font type can be selected by using the combo box or the font face menu.
  - The font style can be selected by using the check boxes or the font style menu.
  - The font size can be selected by using the radio buttons or the slider.
- In the middle of the GUI, the size label always displays the current font size.
- Make the combo box editable. When a new font is entered, it should be stored in the combo box and menu drop down lists.
- When the font is modified, the width of the window must adapt itself automatically (hint: use `Stage.sizeToScene`)
- All inputs must always remain **synchronized**. For example:
  - If the style "italic" is selected in the menus, the corresponding check box must programmatically be selected, and conversely;
  - If the size "large" is selected with a button, the slider must be set to the corresponding value, and conversely; etc.

Where appropriate, try to use binding operations.

In Moodle, you will find the image ("`BigJava.jpg`") and a simplified basic structure for your program ("`FontViewer.java`") that you can use if you want:

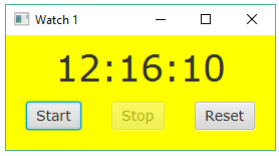
https://moodle.bfh.ch/mod/folder/view.php?id=631733



Exercise 2 – MVC

Extend the counter example that we saw during the lessons on the MVC pattern as follows:

- Instead of a single counter, create watches (at least two of them) with a HH:MM:SS display that can be started, stopped and reset. When they are reset, the watches must display the current system time; when they are started, they count the seconds. Each watch must be contained in a separate window. Example:



- To control the watches, define at least two control units with three buttons (start, stop, reset). Each control unit must control all the watches (i.e. when the start button of one control unit is clicked, all watches start; when one of the stop buttons is clicked, all watches stop, and so on). Build one of your control units together with the watch display (i.e. in the same window) and one in a separate window (therefore, you should have at least three windows in your application). Make various layouts.
- When the watches have been started, the start buttons must be disabled until the watches are stopped. Similarly, when the watches have been stopped, the stop buttons must be disabled until they are restarted. In one of the control unit, display a corresponding message “Watches are running”, bzw. “Watches are not running”.
- Where appropriate, use binding operations.

Use object serialization to control the behavior of the application by terminating and restarting:

- When any of the windows is closed, the application must terminate.
- When the application restarts, the watches must not be reset, but conserve the value they had when the application terminated.
- Similarly, the state of the GUI should not change when the application terminates and restarts, i.e.:
    - Buttons that have been disabled by termination should still be disabled by restart.
    - If the watches were incrementing by termination, they must be further incrementing, as soon as the application restarts, without any action needed on the start buttons.

Exercise 3: Read & Write Files

For this exercise, you are given in Moodle two disk files, `ByteAccounts.dat` and `DataAccounts.dat`. These files contain data records for bank accounts in different formats. Every record consists of the account number (integer), the owner (text), the balance (floating-point number) and the creation date (text formatted as dd.mm.yyyy). Both files have been generated by the program `BankDataGenerator` that is also given, together with the input file `Names.txt`. See:

`https://moodle.bfh.ch/mod/folder/view.php?id=631759`

Understand the formats of `ByteAccounts.dat` and `DataAccounts.dat` by reading the `BankGenerator` program and then implement a `Bank` class that contains the following methods:

- `readByteAccounts:` this method reads the bank account data from the file `ByteAccounts.dat` and stores them in a text file `Accounts1.txt` that has the following format (one line per account):

  ```
  #: 1501 \ Nora Logan \ Balance: 50769.47 \ Creation date: 04-12-2001
  #: 2102 \ Brenna Wiggins \ Balance: 17013.18 \ Creation date: 02-05-2017
  ...
  TOTAL: 15259147.99
  ```

  At the end of the file, append an extra-line that contains the total balance of all accounts.

- `readDataAccounts:` this method reads the bank account data from the file `DataAccounts.dat` and stores them in a text file `Accounts2.txt` that has the same format as `Accounts1.txt`. (Note: both text files should have the same content).

- `sortBalances:` this method reads the accounts from one of the files `Accounts1.txt` or `Accounts2.txt` in an array list, sorts that array list in ascending order according to the account balance values, and stores the sorted accounts in a new text file. Also append an extra-line with the total balances.

- `sortDates:` this method is similar to `sortBalances` but sorts the array list according to the account creation dates.

Write a main program that instantiates the bank, and invokes the various bank methods. Implement a reasonable error handling that throws an exception if one of the initial input files (`ByteAccounts.dat` or `DataAccounts.dat`) is not properly formatted, and give the user a chance to select another file.