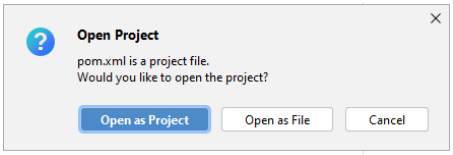
VOP Exam June 06, 2023

This exam contains 3 programming exercises, which weighs 100% of the final grade in the VOP course. We have provided a *ZIP file* called *VOPExam2023.zip*. *UNZIP* this file. You will find 1) a pdf file "**VOPExam23-instructions**" and 2) a maven project "**VOPExam23-Exercise**". Open "*VOPExam23-instructions*" to see the instructions for completing the exam exercises. Open the maven project "*VOPExam23-Exercise*" to see the packages containing the code snippets of the programming exercises. To open the project, go to *File -> open* in IntelliJ and navigate to the "pom.xml" file in the unzipped project folder. After clicking "pom.xml" file, select the option "open as project". It will open the project in IntelliJ.



Submission

At the end of the exam, the solution must be handed over to the Digital Exam:

- In IntelliJ go to "*File -> Export ->Project to Zip File…*"
- Name the zip file after your SDU Username such as "abcd17.zip"
- Upload the zip file
- For your own security, download the zip file and unzip it to verify that you uploaded the right file

Hint: *Take some time to read through the set of exercises before you start working on the solutions.*

Exercise 1: 40 points

This exercise is based on multithreading and thread synchronization. In this exercise, you need to write a singleton logger class for a bank. The logger class should register all the withdraws, deposits and transfers in a single file "log.txt", such that a thread 1 logs withdraw in the text file, thread 2 logs deposit and thread 3 logs transfer in the text file. In this exercise, you need to avoid race condition, such that when one thread is accessing the Logger object, other threads will wait to access the same object at a time until their turn comes.

In this exercise, we have provided 2 classes called "`ThreadDemo.java`" and "`Logger.java`" in the package *multithreading*. `ThreadDemo.java` is fully implemented and responsible for creating and executing the threads. `Logger.java` is partially implemented. It is a singleton class, which restricts the instantiation of a class to **one** object.

Implementation of Logger (Singleton) class

- A private static variable `logger` of type Logger is already declared.
- A public static method `getInstance()` is already implemented to provide the global point of access to the Logger object.

Follow the instructions below to complete the implementation of Logger class. (**Hint:** Remember to use the right access modifiers for variables, constructor and methods). Make sure to catch the relevant exceptions that can be thrown.

- Declare two variables `logFile`(String), `writer`(PrintWriter/FileWriter).
- Create a no-arg (0 argument) constructor to initialize the variable `logFile`, i.e., assign "log.txt" to it. (**Hint:** pay attention to the access modifier of the constructor, as it's a singleton class)
- Implement `logWithdraw`(String `account`, double `amount`) method.
    - Use implicit/explicit thread synchronization mechanism to avoid race condition.
    - Create a writer object to open the file `logFile` and assign it to the variable `writer`(declared above). (**Hint:** remember to open the file in an *append* mode)
    - Write the **account** number and withdraw **amount** in the text file.
      for e.g., `WITHDRAW (002): 500.0$`
- Implement `logDeposit`(String `account`, double `amount`) method.

- Use implicit/explicit thread synchronization mechanism to avoid race condition.
- Create a writer object to open a file `logFile` and assign it to the variable `writer`. (**Hint**: remember to open the file in an *append* mode)
- Write the **account** number and deposit **amount** in the file.
  for e.g., `DEPOSIT (001): 200.0$`

- Implement `logTransfer(`String `account`, String `transferAccount`, double `amount)` method.
  - Use implicit/explicit thread synchronization mechanism to avoid race condition.
  - Create a writer object to open a file `logFile` and assign it to the variable `writer`. (**Hint**: remember to open the file in an *append* mode)
  - Write the **account** number, **transferAccoun**t number and **amount** in the file.
    for e.g., `TRANSFER (005->001): 1000.0$`

Implementation of WithdrawTask, DepositTask and TransferTask

Create `WithdrawTask.java`, `DepositTask.java` and `TransferTask.java` as follows,

- Create a class `WithdrawTask.java` with the signature `public class WithdrawTask implements Runnable` in the *multithreading* package.
- Declare 3 variables `account`(String), `amount`(double), `logger`(Logger)
- Create a three-argument constructor to initialize the variables `account`, `amount` and `logger`.
- Implement the `run()` method that invokes `logWithdraw()` method of `logger` object.

- Create another class `DepositTask.java` with the signature `public class DepositTask implements Runnable` in the *multithreading* package.
- Declare 3 variables `account`(String), `amount`(double), `logger`(Logger)
- Create a three-argument constructor to initialize the variables `account`, `amount` and `logger`.
- Implement the `run()` method that invokes `logDeposit()` method of `logger` object.

- Create another class `TransferTask.java` with the signature `public class TransferTask implements Runnable` in the *multithreading* package.
- Declare 4 variables `account`(String), `transferAccount`(String), `amount`(double), `logger`(Logger)
- Create a four-argument constructor to initialize the variables `account`, `transferAccount`, `amount` and `logger`.
- Implement the `run()` method that invokes `logTransfer()` method of `logger` object.
- Test the implementation by running the "`ThreadDemo.java`", uncomment the code in the `main()` method. **Example** of correct output (**log.txt**). Remember, the order of deposit, transfer and withdraw statements in the file could be different depending on which thread acquired the lock first.

```
1  DEPOSIT (001): 200.0$
2  TRANSFER (005->001): 1000.0$
3  WITHDRAW (002): 500.0$
```

Exercise 2 — 5 points

Implementation of SumOfAllNumbers

In this exercise, you are supposed to compute sum of natural numbers using recursive approach. To do this, we have provided a class called "**SumOfAllNumbers**.java" in the package recursion. SumOfAllNumbers.java is **partially** implemented and responsible for calculating a sum of a series of natural numbers from **1** to **50,** using both iterative and recursive approach**.**

- A `iterativeSum()` method is already implemented to compute a sum of a series of natural numbers using **iterative** approach.
- A `main()` method is already implemented.

Complete the implementation of the class.

- Implement `recursiveSum()` method to compute a sum of a series of natural numbers using **recursive** approach.
- Run "SumOfAllNumbers.java" to test your implementation.

Example of correct output

```
Iterative Sum=1275
Recursive Sum=1275
```

Exercise 3: 55 points

Tourism is important because it can contribute significantly to a country's economy by creating jobs, generating income, and promoting the development of infrastructure and services. This exercise employs a dataset, which contains information about **international tourism receipts** for various countries. International tourism receipts refer to the expenditures made by international visitors on their trips to a country, including accommodation, food and beverage, transportation, and other tourism-related expenses. In this regard, a *comma-separated file* called "***tourism-receipts.csv"*** is provided, which comprises information on tourism receipts for over 100 countries from the year 1995 to 2020.

- Each row in the file ("***tourism-receipts.csv***") represents information about a particular country, comprising 4 fields (in blue) as follows:
  name (name of the country), code (code of the country), year (year the data was recorded), value_$ (total amount of international tourism receipts in US dollars)
- These fields can be seen as the first entry in the "***tourism-receipts.csv***" file.
- This exercise is divided into four sub-exercises, where you need to read CSV file and sort the data with respect to different attributes and create a simple GUI interface for viewing tourism data according to their countries.

Exercise 3a: Tourism implements Comparable <Tourism> 10 points

In this exercise, you will sort data using Comparable interface. We have provided a class called "`Tourism.java`" in the package *tourism_info*. An instance/object of the class should represent one entry in the tourism-receipts.*csv* file. Complete the implementation of the class. (**Hint:** Remember to use the right access modifiers for variables, constructor and methods). Make sure to catch the relevant exceptions that can be thrown.

- Declare 4 variables for `name`(String), `code`(String), `year`(int), `value`(double).
- Create a 4 argument constructor to initialize 4 variables.
- Create 4 `Getter` methods for retrieving the values of all the 4 variables. A `Getter` method should return the value of the variable. An example of a `Getter` method is given below for `name(String)`:

```
public String getName() {
    return name;
}
```

- Implement a `compareTo()` method, i.e., compare the `value` of two `Tourism` objects. (**Hint:** Use the corresponding `Getter` methods to sort the `value` in ascending order (Low to High).
- A `toString()` method is already implemented but commented out. Uncomment this code and ensure that the called `Getter` methods in the `toString()` method corresponds to the ones you have created.

- A `main()` method is already implemented but the lines of code in the method are commented out. Uncomment the code under **Exercise 3a** only to test your implementation so far.

Example of correct output

```
Sorting based on Value
[Afghanistan         AFG         2017        16000000,00
, Morocco        MAR         2018        9520000000,00
, Canada         CAN         2007        17962000000,00
]
```

Exercise 3b: Sorting with Comparator 10 points

In this exercise, you will sort data using Comparator interface. Follow the instructions below.

- Create a class `YearComparator.java` with the signature `public class YearComparator implements Comparator<Tourism>` in the *tourism_info* package.
- Implement the `compare()` method to compare two `Tourism` objects by their `year` and if two objects have the same `year`, they should be compared by their `value`. (**Hint**: Remember to use the corresponding `Getter` methods in the `Tourism` object).
- Create another class `NameLengthComparator.java` with the signature `public class NameLengthComparator implements Comparator<Tourism>` in the *tourism_info* package.
- Implement the `compare()` method to compare two `Tourism` objects by their `name length` and if two objects have the same `name length`, they should be compared by their `value`.. (**Hint**: Remember to use the corresponding `Getter` methods in the `Tourism` object).
- The `main()` method of the Tourism class is already implemented, uncomment the code under **Exercise 3b** to test your implementation.

Example of correct output:

```
Sorting based on Year
[Canada          CAN         2007        17962000000,00
, Afghanistan        AFG         2017        16000000,00
, Morocco        MAR         2018        9520000000,00
]
Sorting based on Name Length
[Canada          CAN         2007        17962000000,00
, Morocco        MAR         2018        9520000000,00
, Afghanistan        AFG         2017        16000000,00
]
```

Exercise 3c: Implementation of ReadCSV 15 points

In this exercise, you will reading a comma separated file. Follow the instructions below. For this exercise, we have provided a class ReadCSV.java in the tourism_info package. The ReadCSV.java is partially implemented. We have provided the following:

- A private variable `file` of the type `File`, is already declared.
- A private variable `map` of the type `map<String, Set<Tourism>>`, is already declared.
- A constructor `ReadCSV(String fileName)` that instantiates the declared `map` and `file` variables, is already created.
- An overloaded constructor with the signature `public ReadCSV(File file)`, is already implemented.
- A `Getter` method called `getMap()` that returns the instantiated `map` variable, is already created.

Complete the `ReadCSV.java` class Implementation as follows:

- Implement the `readFile()` method in the class to read each line in the file "*tourism-receipts.csv*". Remember that this file is comma separated "," and relevant fields to be read are provided in the introduction to this exercise. Remember to skip the first line in the file - as it is an information line.
  - Ensure that you enclose the input stream in a `try – catch` clause and close your input stream after use.
  - Make sure to catch the relevant exceptions that can be thrown.
  - Read each line in the *tourism-receipts.csv* file and parse the relevant fields of the line into their appropriate types to create a new `Toursim` object.
  - Use the relevant `Getter` method of the `Tourism` object to get the country `name` parameter and use this parameter to check if the country `name` exists in the `map` object. See an example for inspiration below:

```
this.getMap().containsKey(tourism.getName())
```

  - If the country `name` does not exist in the `map`, create a new sorted `Set`, add the new `Tourism` object to the new `Set` and put the country `name` and its corresponding `Set` object in the `map`.
  - If the country `name` already exists, simply add the newly created `Tourism` object to the sorted `set` of `Tourism` belonging to that particular country `name`.
- Test your implementations, by executing the `main()` method of the `ReadCSV.java`.

Example of correct output showing the beginning and ending the of execution (separated by "---"):

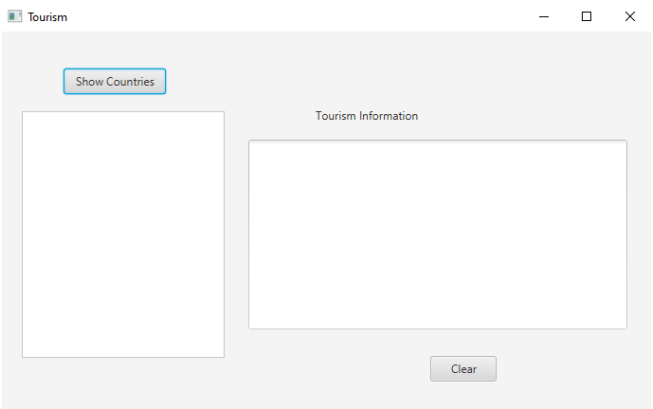
```
{Zimbabwe=[Zimbabwe          ZWE         2003        61000000,00
, Zimbabwe       ZWE         2020        66000000,00
, Zimbabwe       ZWE         2002        76000000,00
, Zimbabwe       ZWE         2001        81000000,00

 ---
 ---
 , Afghanistan       AFG         2011        165000000,00
 , Afghanistan       AFG         2012        167000000,00
 , Afghanistan       AFG         2013        179000000,00
 ]}
```

Exercise 3d: Implementation of primary.fxml and PrimaryController 20 points

In this exercise, we have provided two classes namely *App.java*, *PrimaryController.java* and an *FXML* document called "*primary.fxml*". The two classes can be found in the *org.example.vopexam2023* package, while the *primary.fxml* can be found in the *org.example.vopexam2023* folder under the *resources* folder. The *PrimaryController.java* is the controller class for the *primary.fxml* document. Both files will be used to implement this exercise.

Create the layout that can be seen below in *primary.fxml* document with the following items:

- A Button called "Show Countries"and associate it with a `showCountries()` "onAction" handler.
- A `ListView` with *fx:id* `lsCountries` and associate it with `showTourInfo()` "OnMouseClicked" handler. This `ListView` should show the list of the countries in the sorted `Map` of `ReadCSV.java`.
- A static `Label` that shows the text "Tourism Information".



- A `TextArea` with *fx:id* `txtTourInfo` for showing the Toursim Information of a selected country in the `ListView`.
- A Button called "Clear"and associate it with a with a `clearAction()` "onAction" handler.

Complete the implementation of the `PrimaryController.java` class as follows:

- The "@FXML" annotated attributes for ListView. i.e., `lsCountries` and TextArea, i.e., `txtTourInfo` are already created to make them accessible within your "PrimaryController" class.
- A private variable of the type `File` called `selectedFile` is already declared.
- A private variable of the type `ReadCSV` called `readCSV` is already declared.
- We have declared the `showCountries()`, `showTourInfo()`, and `clearAction()` event handlers.

Implement the methods in the PrimaryController.java as follows:

- Implement the `ShowCountries()` method to show the list of Countries in the ListView, i.e., `lsCountries`
    - selectedFile variable is already instantiated.
    - Instantiate `readCSV` and pass `selectedFile` as an argument.
    - Invoke `readFile()` method using `readCSV` object.
    - Set the `ListView` with the `keySet` from the `map`. Use the following code as inspiration.

```
lsCountries.setItems(FXCollections.observableArrayList(readCSV.getMap().keySet()));
```

- Implement the `showTourInfo()` so that it displays **only** the Tourism Information of the country that is selected from the `ListView` to the `TextArea`. Implement this method as follows:

    - Clear the `TextArea`
    - Get the selected item in the `ListView` object `lsCountries`, and assign it to a variable of type `String` as follows: (Hint: Remember to cast the selected item to `String` before assigning to the String variable. Use the following code as inspiration.

```
String country=lsCountries.getSelectionModel().getSelectedItem()
```

    - Create an instance of `TreeSet` of type `Tourism` called `"items"`.
    - Get the Toursim Information of the selected `country` by using the `getMap()` method of `readCSV` object, and add all the information to `"items"` (`TreeSet` object)
    - Iterate over `"items"` and append text to `TextArea`, i.e., `txtTourInfo`.

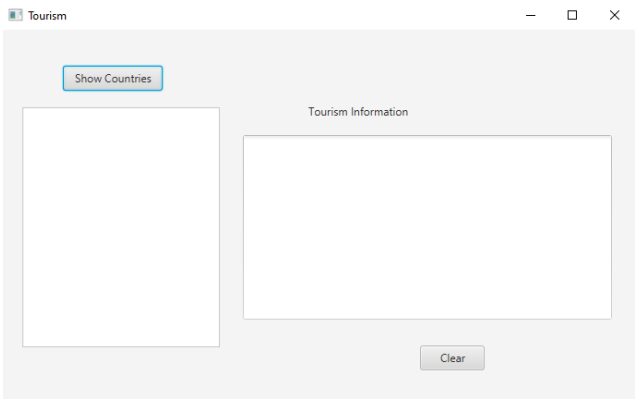
- Implement the `clearAction()` to clear the `TextArea` and to scroll, focus and select the **first** item in the `listView`
    - Clear the `TextArea`
    - Scroll, focus and select the **first** item in the`ListView;` An example for inspiration is given below;

```
lsCountries.scrollTo(0);
lsCountries.getFocusModel().focus(0);
lsCountries.getSelectionModel().select(0);
```

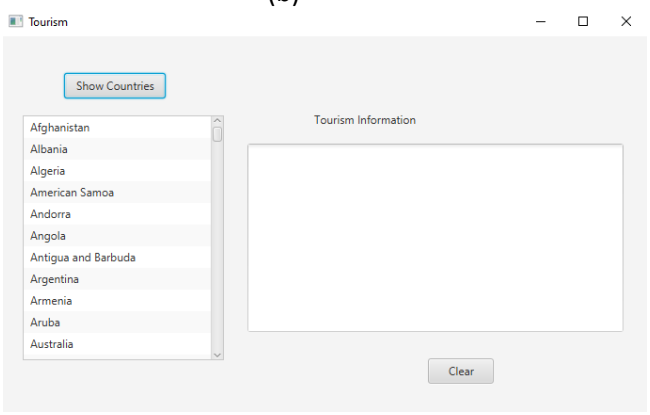
- Test your implementation by running the `App.java` class in the *org.example.vopexam2023* package. If you click on the "Show Countries" button, you should see a list of countries in the ListView. Select a country from the `ListView` (for example *Algeria*).

Example of correct outputs in Figure (a) – (d). Figure (a) shows the result when "App.java" is run. Figure (b) shows the result when "Show Countries" button is clicked. Figure (c) shows the result when "*Algeria*" is selected from the list of countries in the `ListView`. Figure (d) shows the result when Clear button is clicked.

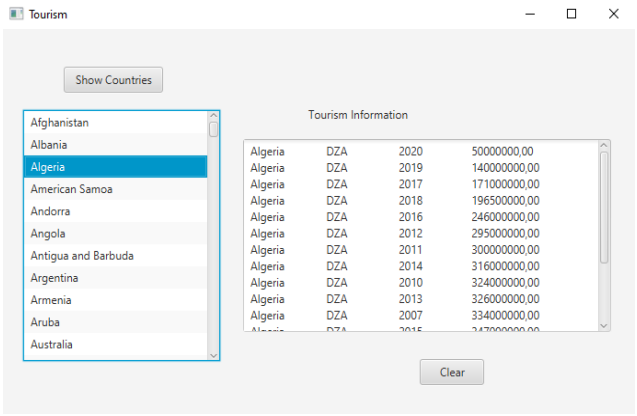
(a)


Tourism
Show Countries
Tourism Information
Clear

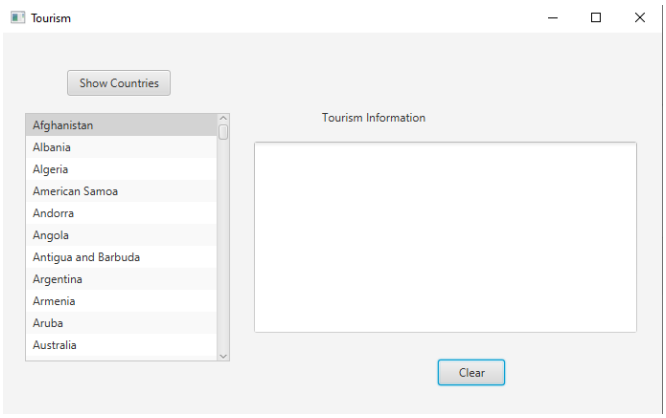
(b)


Tourism
Show Countries
Afghanistan
Albania
Algeria
American Samoa
Andorra
Angola
Antigua and Barbuda
Argentina
Armenia
Aruba
Australia
Tourism Information
Clear

(c)


Tourism
Show Countries
Afghanistan
Albania
Algeria
American Samoa
Andorra
Angola
Antigua and Barbuda
Argentina
Armenia
Aruba
Australia
Tourism Information
Algeria DZA 2020 50000000,00
Algeria DZA 2019 140000000,00
Algeria DZA 2017 171000000,00
Algeria DZA 2018 196500000,00
Algeria DZA 2016 246000000,00
Algeria DZA 2012 295000000,00
Algeria DZA 2011 300000000,00
Algeria DZA 2014 316000000,00
Algeria DZA 2010 324000000,00
Algeria DZA 2013 326000000,00
Algeria DZA 2007 334000000,00
Clear

(d)


Tourism
Show Countries
Afghanistan
Albania
Algeria
American Samoa
Andorra
Angola
Antigua and Barbuda
Argentina
Armenia
Aruba
Australia
Tourism Information
Clear