Lab 1 : Introduction to Scala

The goal of this first lab work is to get students started with Scala's syntax, for which an introduction was made during the first lesson.

The main notions that will be put in practice are:

- Variable definition
- Method definition & function definition
- Branching and looping
- String formatting
- I/O

We will begin by testing some simple commands in a shell (Scala's REPL, or "interpreter") and implementing some basic interaction with the user. Later, we will write the commands in a file, called a **script**, that contains a reusable sequence of code that can be launched using the **scala** command.

The theme of this lab work (and the next one) will be the construction of a very basic virtual assistant (no voice recognition, no fancy features) that will:

- take commands from the user;
- do some computation;
- display some output or write it to files.

Let us call it PrimA (for "**Prim**itive **A**ssistant").

Warm-up

Open either the basic Scala REPL or Ammonite (for syntax highlighting).

Follow the installation tutorial for Scala/Ammonite if you don't have it yet.

NB ! Do not hesitate to use comments for you to remember what your code does (and for me to better understand your code!).

Exercise 0. Print a `"Hello, world!"` greeting inside the shell.

Exercise 1. Define a variable **`name`** and let your name be its value. Print a `"Hello, world!"` greeting, but use the **`name`** variable instead of `world`: use string formatting. Try using the **`val`** and **`var`** keywords to create the variable and, for each case, try changing **`name`**'s value after its first initialization : see what happens…

Exercise 2. Define a variable **`inputName`** and let it be initialized with a value prompted from the user (hint: use `scala.io.StdIn`, seen in the 1st lesson's quickstart). Print the previous greeting by using **`inputName`** instead of **`name`**.

Exercise 3. Define a method **`greetUser`** that takes a `String` argument and uses it as a name to print the previous greeting.

Exercise 4. Define a function **`greetUserFun`** that takes a `String` argument and uses it as a name to return a greeting `String` such as `"Hello, Bob !"`. *The function should return it, not print it!* Note: A method's or function's result value is the last statement mentioned it its body.

Exercise 5. Define a method **`greetWithCustomGreeting`** that takes 2 arguments, a name of type `String` and a function that uses a `String` to return a new `String` (: what will the type of the function be?). **`greetWithCustomGreeting`** will print the greeting generated by its second argument that will use the first argument to generate a greeting. Define a function **`greetUserFun2`** that will generate some other form of greeting (such as **"`Welcome, Bob !`"**), then use **`greetWithCustomGreeting`** twice, using different functions as arguments.

Exercise 6. Define a variable **`guests`** and initialize it with a list of names, such as `Alice, Bob, Sam`. Use a **`for` loop** to print a greeting for each guest using **`greetWithCustomGreeting`** and **`greetUserFun`**. Optional : try doing the same with **`foreach`** :) .

Exercise 7. Manually create a `guests.txt` file containing names (1 name per line), as follows:

Alice
Bob
Sam

Define a variable **`guestsFromFile`** and initialize it with the lines read from the `guests.txt` file (hint: use `scala.io.Source`, seen in the 1st lesson's quickstart). Use the file's absolute path to avoid "`File not found`"-like errors.

We shall now use a **`while` loop** in the following steps to print a greeting for each guest using **`greetWithCustomGreeting`** and alternating between **`greetUserFun`** and **`greetUserFun2`**. The goal is to greet each second guest with **`greetUserFun2`**, as follows:

Hello, Alice !
Welcome, Bob !
Hello, Sam !

1. Define a variable **`indexOfGuestToGreet`** and initialize it with 0. The variable will be used to keep count of the number of already greeted guests, but ALSO as the index (or position) of the name to retrieve from the **`guestsFromFile`** list. What keyword will be used for the variable: **`var`** or **`val`** ?

2. Use a **`while` loop** to greet the guests in **`guestsFromFile`** using **`greetWithCustomGreeting`** and **`greetUserFun`**. The loop will stop after all guests are greeted: use **`indexOfGuestToGreet`** by comparing it to **`guestsFromFile`** list's length.

   Hint : define guestsFromFile as follows :

   ```
   val guestsFromFile = f.getLines().toList // use toList to transform the iterator into a list, as mentioned by one of your peers
   ```

   then use **`guestsFromFile.length`**.

   As the list's indexing begins with 0, what will be last name's index in terms of **`guestsFromFile`**'s length?

3. The loop created in 2. greets every name with the same formula. In order to alternate between 2 formulas, "Hello" and "Welcome" (**`greetUserFun`** and **`greetUserFun2`**), we will use branching with the **`if…else`** construct. Greet every name with **`greetUserFun`** if their index is odd, and **`greetUserFun2`** otherwise (hint: use **`%`** for "modulo").

Exercise 8. Using the looping keyword of your choice, we will follow the next steps to generate a `greetedGuests.txt` file that will contain the output displayed in Exercise 7:

Hello, Alice !
Welcome, Bob !
Hello, Sam !

As **`greetWithCustomGreeting`** prints the output inside the shell, we will not use it. Instead, we will define a method that will generate a list of greeted guests that will then be written to a file.

1. Define a **`generateGreetingsList`** method that takes a list of `String` and 2 functions that uses a `String` to return a new `String`. The method will adapt the loop from Exercise 7 to prepend new `String` elements to an initially empty list, then reverse it and return the resulting list.

   Hints:

   - to create an initial empty list of `String`, use

     ```
     var listName: List[String] = Nil // or List() instead of Nil
     ```

     . *`Nil` is an empty list in Scala.*

     Note: As lists have a constant length and, moreover, are immutable, we will use **`var`** in order to update the variable **`listName`** with a resulting list.

   - to prepend a value to a list, use `::`

     ```
     “Some string” :: listName
     ```

     Note:

     `listName :: “Some other string”` will not work !!!

     `::` is applied to the list and not to the `String`.

     We will see why during following lab works.

   - to reverse a list, use

     ```
     listName.reverse
     ```

2. Use **`generateGreetingsList`** with **`guestsFromFile`**, **`greetUserFun`** and **`greetUserFun2`** as arguments. Store the result in a variable **`generatedGreetings`**. Write the contents of **`generatedGreetings`** to the `greetedGuests.txt` file.
   Hint: use `java.io`, seen in the 1st lesson’s quickstart.

Exercise 9. In this exercise, instead of using the REPL, we will write a script and execute it using the `scala` command. You can close the REPL by writing **`:quit`** (then press **`<Enter>`**).

1. Create a file named `helloUser.scala` . This file will be our script.
   Open the file, then type in:

   ```
   println(“Hello, user!”)
   ```

   Note: Rewrite the code instead of copying to avoid errors due to bad pasting.

Save the file, then run the script following command in a shell such as **`cmd.exe`** or **`bash`** (replace `/path/to/` with the true path):

```
scala /path/to/helloUser.scala
```

2. Let's customize the message using a name passed as an argument instead of **"user"**, such as in:

```
scala /path/to/helloUser.scala John
```

which should print

```
Hello, John!
```

When running a script, command-line arguments are captured within an array called **`args`**, in the order in which they appear separated by spaces in the command-line. Consequently, to access the first argument, use **`args(0)`**.

Here is what our new script `helloUser.scala` will look like:

```
println(s"Hello, ${args(0)}!")
```

Try executing the new version of the script using the command mentioned at the beginning of **2.** .

PrimA: Part 1

At the end of Part 1, each student will have a Scala script that can take an input file as a parameter and generate an output file. It will be used to check if the code works as expected against a test input file.

During the following exercises, it is recommended to write variable, function and method definitions in the file that will become your script, and test the methods during development by copying them into the REPL.

Exercise 10. PrimA will need some knowledge about its user. Start with creating a `mutable.Map[String, String]` containing 3 entries with empty strings as values:

- "name" -> ""
- "surname" -> ""
- "date of birth" -> ""

Use the documentation at https://docs.scala-lang.org/overviews/collections-2.13/maps.html or the 1st lesson's quickstart for the syntax.

Exercise 11. Define a method that takes 4 arguments:

- A `mutable.Map[String, String]` to which we will pass the mutable map
- 3 `String` to which we will pass the new values for **`name`**, **`surname`**, **`date of birth`**.

The method will update any entry of its 1st argument for which the new value (given in the following 3 arguments) is different from an empty string.

The method's result type will be `Unit` (it will return nothing).

Exercise 12. Manually create a `userInfo.txt` file containing user information (1 field per line, the field's name and value will be separated by a column **":"**), as follows:

name:John
surname:Smith
date of birth:20000101

Put in the values you wish, but the keys must remain unchanged. The "date of birth"'s format should be: 4 digits for the year, 2 digits for the month and 2 digits for the day, all glued together.

Define a method that takes 1 argument:

- A `String` to which we will pass the path to `userInfo.txt`

The method will read the file and use the method from Exercise 11 to update the entries of the map containing the user information.

Hint: you can use previously defined methods inside the body of the new method, as in:

```
def foo(): String = {
      "hello"
}
def foo2(): String = {
      foo() + ", world"
}

foo2()
```

Hint 2: you can use the `split` method to split a string based on a given substring.

Example:

```
"Hello, world".split(",")
```

will result in

```
Array[String]("Hello", " world")
```

The method's result type will be `Unit` (it will return nothing).

Exercise 13. Create a second `mutable.Map[String, String]` that will contain scheduled activities for the user. The map will be empty at first: how do you create an empty mutable map ? (hint: one way of doing it is similar to creating an empty `List`, except the value should be given as `Map()` instead of `Nil` or `List()`).

The map will later contain entries such as the following example:

"20220404 080000" -> "Scala course", "20220404 130000" -> "Lunch"

Exercise 14. Create a method that takes 1 argument :

- A `mutable.Map[String, String]` to which we will pass the mutable map containing scheduled activities.
  The method will add scheduled activities by asking the user to write the time and subject in the command line.

Exercise 15. Manually create a `userActivities.txt` file containing activities to schedule (1 field per line, the field's name and value will be separated by a column **":"**), as follows:

20220404 080000:Scala course
20220404 130000:Lunch

Follow the same principle as in Exercice 11 for the date's / time's format.

Define a method that takes 1 argument:

- A `String` to which we will pass the path to `userActivities.txt`

The method will read the file and add the new entries to the user activities map, following the same principle as in Exercice 11.

Exercise 16. Define a method that takes 2 arguments:

- A `mutable.Map[String, String]` to which we will pass the map containing user activities
- A `String` to which we will pass a date (same format as before, i.e., 4 digits for the years, 2 digits for the month and 2 digits for the day, all glued together).

The method will go through the elements of the map given through its first argument and compare contents of each element to the second argument in order to retrieve activities for a given date. The filtered activities will be returned as a new `mutable.Map[String, String]`.

Exercise 17. Define a method that takes 3 arguments:

- A `mutable.Map[String, String]` to which we will pass the map containing user information
- A `mutable.Map[String, String]` to which we will pass the map containing user activities
- A `String` to which we will pass a date (same format as before, i.e., 4 digits for the years, 2 digits for the month and 2 digits for the day, all glued together).

The method will return a `List[String]` that will contain:

- a greeting for the user (using their name and surname): the greeting will be a congratulation if the date passed through the 3rd argument will be equal to the user's date of birth, otherwise it will be the generic greeting of your choice;
- the scheduled activities for the date given as the 3rd argument.

Exercise 18. Define a method that take a `List[String]` and write its contents to a file. What arguments will you need?

Exercise 19. We will use most of the previously defined methods to create a menu that will ask the user what he wants to do. The menu will propose the following:

1. Create the user from a given file (prompt for the file path).
2. Update the user from the command line (prompt for the name, surname, date of birth).
3. Add user activities from a given file (prompt for the file path).
4. Add a user activity from the command line (prompt for the date-time and subject).

5. Greet and show activities for a given day (prompt for date).
6. Write a greeting and the activities for a given day in a file (prompt for the date and the file path).
7. Exit.

The user will select a number and write it in the command line, after which the script will execute the selected action. In order to show the menu repeatedly after some action is done, we could use a **`while` loop** that shows the menu as long as the input is different from **`7`**.

To fulfill this task, we could develop new methods that prompt for user input and call the methods from previous exercises to update the map variables and/or print/write the greeting and activities.

Exercise 20. Endgame.

Let's re-arrange the code developed in the previous exercises in the following manner:

First, variable definition and initialization: here, we will define and initialize the user information and activities maps. The **`mutable`** package will be imported here.

Second, function and method definitions: here, we will define the methods used by PrimA's menu (the **`while` loop**). Make sure the definitions are given in the correct order: in this script, a method's definition should precede its use, e.g., by another method.

Third, the **`while` loop**.

The code will go into a `PrimA_part1.scala` script that users will be able to launch using the following command:

```
scala PrimA_part1.scala
```

Executing this command should display the menu from which the user can choose actions to execute.

Here is what the script should look like:

```
//import mutable
import scala.collection.mutable
//define maps
val userInfo = …
val userActivities = …
…
//define methods (you can pick your own method names)
def updateUserInfo( … ) … {
  …
}
def addActivity( … ) … {
  …
}
…
```

```
//while loop
var menuChoice = …
while ( … ) {
  …
}

//end of script
```

After making sure the script works as expected, we will add the last functionality to our script. Instead of having an interactive menu, the user should also be able to call the script with some arguments to only execute option **6** of the menu once.

The user will provide the following arguments:

- the paths to 2 input files (user information and user activities)
- a date using the previously seen format, for which they want to display activities
- the path to an output file, in which the greeting and activities will be written.

The command will look as follows (all in one line, arguments separated by spaces):

```
scala PrimA_part1.scala /path/to/userInfo.txt /path/to/userActivities.txt 20220404
/path/to/greetingAndActivities.txt
```

The behaviour can be achieved by adding an **`if…else`** statement around the **`while` loop**, conditioned by the length of the **`args`** variable available in the script during its execution: if arguments were passed to the script, then execute the same code as in option **6**, otherwise, run the loop. Here is an example:

```
//while loop
if (...) {
      // some code or method from option 6
      // …
} else {
      var menuChoice = …
      while ( … ) {
        …
      }

}
```

Brain teasers

Exercise 21. Define a method that checks that a given **`String`** passes the requirements to be a password.

The password should contain :

- At least 9 letters
- At least 1 capital letter
- At least 1 lower case letter
- At least **m** different sequences of numbers (**m** will be one of the method's arguments). Each sequence will contain 2 or more non-repeating numbers, as in 924gjr374ze.
- At least **n** special characters from a list **l** (**n** and **l** will also be the method's arguments) in non-adjacent positions of the password

Any other character will not be accepted.

Here are some examples of bad passwords : (given m = 2, n = 3, l = ("&", "?", "@")) :

- 9Fejg (rule 1)
- **99**3fE@z4?4 (rule 4)
- **123**fE@z**123**?e& (rule 4)
- 123fE@z124**?&**e (rule 5)

The method will return a **`Boolean (true or false)`**.

Exercise 22. Code a 2-player tic-tac-toe game. (If you don't know what tic-tac-toe is, ask the instructor.)

The game should display the state of the board at each round (as in hint 2).
It should ask for input from player 1 and player 2 alternatively (asks with a message).
If an incorrect input is given, or the position is already filled, it should ask for input again (from the same player).
The game stops if the board is filled or there is a winner (a player wins if he fills a row, a column or a diagonal).
It announces the result.
It then asks if players want to play again, and resets the board if needed.

Use the code from a Scala script, **`tictactoe.scala`** .

Hint : how can you encode a matrix ? Also, think about mutability.
Hint 2 : you can use stripMargin on a String.

```
println(
""" o | x | o
  ?-----------
  ?
  ?   | x | x
  ?-----------
  ?
  ? x | o |   """.stripMargin('?'))
```

It will delete all spaces at the left of the character mentioned as **`stripMargin`'s** argument (and the character itself). **NB**! Use single quotes around the character, as it must be of type **`Char`**, that we will see in the following session.

The result will be :

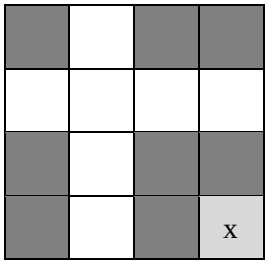
```
 o | x | o
-----------

   | x | x
-----------
```

```
x | o |
```

Exercise 23. Code a simulation inspired by **Conway's Game of Life**.
The idea is having a grid (or matrix) where each cell contains one of the 2 possible states : alive or dead.
The simulation works iteratively, after each cell is given an initial state.
At each iteration, the neighborhood of each cell is considered :

- If a living cell has 2 or 3 living neighbors, it stays alive in the next iteration;
- If a dead cell has 3 living neighbors, it becomes alive in the next iteration ;
- All other living cells die in the next iteration, and all other dead cells stay dead.

The neighborhood of a cell is defined as its 8 adjacent cells. If a cell is on the border of the grid, its neighbors are the cells at the beginning of the opposite border. E.g., if a cell is in the lower-right corner, its lower-right neighbor will be the cell in the upper-left corner of the grid, as in the example (neighbors of x are dark-grey) :



The goal is to let the user choose :

- the size of the grid
- the initial state of the grid
- the number of iterations
- the time T between iterations in milliseconds

Starting with the initial state, the solution will show the grid, then show an updated iteration every T milliseconds, until the number of iterations is reached.

Hint : you can use a grid representation similar to the previous exercise.

Hint 2 : to pause for T milliseconds between iterations, use **`Thread.sleep(T)`**.