



## Lab 1: Basics of Java

### Programming Fundamentals 2

16th February 2021



#### Goals

- ★ Using a simple programming workflow: code, compile and run, commit, push.
- ★ Basics of imperative Java: variables, control structures (conditional and loop), arrays.
- ★ Simple object creation and manipulation: constructors, methods, attributes.
- ★ Input/Output library.

#### Deliverables

1. The code on your Github repository generated by clicking here: <https://classroom.github.com/a/owIo2zvP>
2. A presentation video of 3' minutes on FLIPGRID: <https://flipgrid.com/alb7d419>
  - Use your @student.uni.lu email to connect.
  - Share your screen and comment the Exercise 3 (Connect 4).

#### Planning

- **Round 1:** 18th February 08:00AM
- **Round 2:** 22th February 08:00AM
- **Round 3:** 26th February 08:00AM
- **Deadline:** 1st March 08:00AM
- Note: For the rounds 1–3, you just need to push your code on time on Github to get some feedback. The feedback is given on the page of your repository, tab "Actions" (see demo in class).

## Exercise 1 – Pallet Town

(...where all great adventures start...)

**The following steps are crucial. Please reach out for help if something does not work as intended.**

### 1. Set up Sublime Text.

1. Open Preferences > Settings.
2. The left panel is the default setting, and the right panel is your customized preferences which override the default.
3. Add the following in the right panel:

```
{
  "tab_size": 2,
  "translate_tabs_to_spaces": true,
  "trim_trailing_white_space_on_save": true
}
```

This allows us to indent with two spaces by default.

### 2. Open a shell (see *Chapter 0* for instructions on how to install a shell) and type:

```
mkdir PF2
cd PF2
git clone Put here the link to your Github repository for lab1
cd lab1
subl -n .          # does not work on Windows, see the note below.
```

At this point, you should have an example project opened in Sublime Text.

**Note for Windows users**, you cannot type the command `subl -n .` because you cannot launch Windows program from the terminal. The solution is to open Sublime Text as you would open any software, and then to open the cloned repository. For that, in Sublime Text, click on **File**, then **Open Folder**, and navigate the path `\\wsl$\Ubuntu > home > your name > the repository`. If needed, watch the tutorial installation video for an example on how to open “Linux folder” in Sublime Text.

### 3. You can compile and run the project in the terminal by typing:

```
javac -d target src/lab1/HelloWorld.java
java -cp target lab1.HelloWorld
```

You should see the output of the execution of the program on your screen.

- Each time you modify your program, you need to recompile it by calling `javac`.
- The option `-d target` specify the directory in which the compiler will put the compiled classes. It mirrors the folder structure of `src` automatically, e.g., `target/lab1`.
- The compilation produces a file `HelloWorld.class` that can be executed using `java`.
- The option `-cp target` indicates in which directory to find the classes.

**Pitfall 1: Be careful about `lab1.HelloWorld`, the folder and the class are separated by a dot and not a slash!**

**Pitfall 2: When running the commands `javac` and `java`, you must be at the root of the repository just cloned! If you type `ls`, you should see the `src` directory.**

### 4. Let's now create from scratch a simple Java program.

1. Create a file `src/lab1/MessageToTheWorld.java` in the terminal by typing:

```
touch src/lab1/MessageToTheWorld.java
```

- This file is not known to `git`, thus you need to add it:

```
git add src/lab1/MessageToTheWorld.java
```

It is good practise to add your file in `git` immediately after creating it, so you don't forget it.

- Using Sublime Text, create a simple program that prints a message to the world of yours, *e.g.*,

```
package lab1;

class MessageToTheWorld {
    public static void main(String[] args) {
        System.out.println("My name is <your name>, and I'm gonna be a coding legend.");
    }
}
```

**Pitfall 3: Don't forget to write `package lab1;` at the top of the file. It must match the folder path in which the file is stored (below `src/`). For instance, if you have a file `src/myapp/gui/App.java`, then the file `App.java` must start with `package myapp.gui;`**

**Convention 1: The name of the class must start with an uppercase, and be exactly the same as the name of the file without the `.java` extension.**

Compile and execute this program using the command `java` and `javac` described previously. You should see your message to the world printed on the terminal.

- Now you can commit and push your changes on the `git` repository:

```
git commit -a -m "Completed exercise 1: MessageToTheWorld."
git push
```

- At any time, you can check if some files are *untracked* (not known by `git` because you did not write `git add File.java`) or if some changes are not committed yet using `git status`.
- Awesome! You successfully completed the first exercise, it seems you are now ready for the real deal.

## Exercise 2 – Warming up

You should read and understand the `HelloWorld.java` file to complete the following exercises. You can reuse the code of this file. This is especially useful to know how to read user inputs using `Scanner`.

- Create a file `AdvancedGeometry.java`. Ask the user for the height and width of a rectangle and print it with stars. Example:

```
javac -d target src/lab1/AdvancedGeometry.java
java -cp target lab1.AdvancedGeometry
Enter the height of the rectangle: 2
Enter the width of the rectangle: 3
***
***
```

Hint: use `System.out.print` to print something without a newline.

**Pitfall 4: Don't forget to import the class `Scanner` with `import java.util.Scanner`.**

**Pitfall 5: In order to obtain meaningful feedback from the automated correction tool, you should stick to the output we show here (that also holds for the next exercises).**

The next question is annotated “pro”, that means we look into crucial aspects to make your code more professional and reliable.

- Pro.** Strengthen the previous program by verifying the input of the user and asking again if they are not valid (e.g. negative number, letters, ...).  
To do so, create the following function that you will call from the `main` function:

```
// Ask to the user a positive integer using the message 'question'.
// If the user inputs a wrong answer, it prints the error message 'messageIfError' and ask again.
// It repeats until we obtain a positive integer from the user.
public static int askPositiveInteger(String question, String messageIfError) { ... }
```

Hint: use `scanner.hasNextInt()` to check if the next thing to read is an integer, and `scanner.next()` to “throw away” whatever nonsense the user wrote.

**Convention 2: Variable’s and function’s names must use the ”camelCase” notation. It means that it must start with a lower case, and words are separated by upper case, e.g., `computeSum`, `averageOfGrades`**

**Convention 3: Never trust the user, always verify what is given to you.**

**Convention 4: You must never repeat twice the same code, if you are copy/pasting code around, it probably means you need to create a function.**

3. Create a file `Accumulate.java`. Given an array `numbers`, create an array `accumulate` such that `accumulate[i]` contains the sum of the numbers from 0 to  $i$  in `numbers`. The format is as follows (the compilation and execution commands will be ignored from now on):

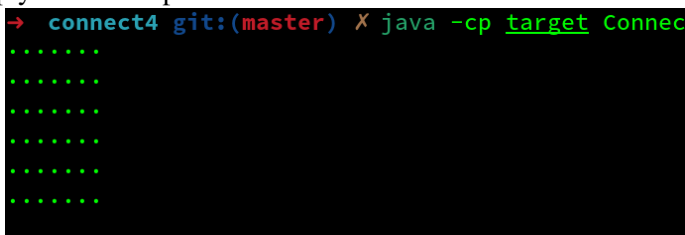
```
Enter the size of the array: 4
Enter the initial array: 2 1 4 8
Accumulated array: 2 3 7 15
```

Hint: To read an array from the user, there is no magic, you need to create an array of size 4, and then read with `scanner.nextInt()` the 4 numbers in a loop to populate the array.

### Exercise 3 – Game development: Connect Four

The goal of this exercise is to develop a command line version of the game *Connect Four*. You can read the rules on Wikipedia: [https://en.wikipedia.org/wiki/Connect\\_Four](https://en.wikipedia.org/wiki/Connect_Four).

As a beginner, it is often difficult to know how to start coding, because the task seems overwhelming and complicated. The key is to **simplify** the task in order to break it into numerous smaller and easier-to-code components. For sure, you do not feel confident to implement the whole game, but what about a program that creates a 2D array representing the empty board and prints it?



```
→ connect4 git:(master) X java -cp target ConnectFour
.....
.....
.....
.....
.....
.....
```

Here we use the dot Unicode character to represent an empty cell in the grid. You can print this character using `System.out.print("\u00B7")` where `00B7` is its Unicode code point. Do you find this task even too complicated? Then you can break it down into a smaller set of tasks again, for instance by only printing a single row of the grid, or even a single dot.

Only beginners code without intermediate steps. When you start coding, you must think about an easy and reachable task, that you think you can complete in 20 minutes. Your *coding workflow* will be to code this task, compile and run the program, possibly debug what you coded, and then commit to git the changes. A task you think you can complete in 20 minutes will probably take longer, so don’t be too eager.

1. The gameplay is very simple and is illustrated in Figure 1. In order to have meaningful feedback from the automated correction tool, an execution of your game should be similar. The name of the main class must be `Connect4.java`. You can have other files with other classes.
2. Before reading the questions that follows, it would be extremely valuable to figure out yourself how you could code this game with very small steps. That is, put yourself in the shoes of a manager who needs to decompose a project into small tasks, so that he can verify the project is progressing well at each step.

3. We decompose the game into a series of intermediate results as follows (that we overly describe to help you!):

1. Create the file `Connect4.java` with a main function printing a welcome message.
2. Create the file `Grid.java` with a two dimensional array as attribute, a constructor, and a `toString()` method. Create an object `Grid` from the main, and print it. Hint on a similar usage of 2D arrays:

```
public class Example {
    private int[][] array2D;
    public Example() {
        array2D = new int[3][2];
    }
    public String toString() {
        String result = "";
        for(int i = 0; i < array2D.length; ++i) {
            for(int j = 0; j < array2D[i].length; ++j) {
                result += array2D[i][j] + " ";
            }
            result += "\n";
        }
        return result;
    }
}
```

And in a main function:

```
public class Main {
    public static void main(String[] args) {
        Example e = new Example();
        System.out.println(e.toString());
        //Note that 'toString' is called automatically in a 'println' so you can simply write:
        System.out.println(e);
    }
}
```

3. Ask the player for the number of the column in which the next disc will be put. You can already check if the user input a correct column number. Feel free to reuse functions of the previous exercises.
  4. Ask the two players in turn their column's number. Put this in a loop. Note: you can press "control + C" to exit a program that does not terminate.
  5. Print the grid each time the user plays. Of course, at this stage, the grid is still empty.
  6. In the class `Grid`, add a method to drop a disc in the grid, for instance `public boolean drop(int x, int color)`. The Boolean indicates whether the disc could be dropped or not in the column `x`; it returns `false` if the column is full.
  7. Call this method in your gaming loop for each player. Ask for a new column position if the column was full.
  8. For now, we do not have winning condition, so the user will play until the grid is full, and won't be able to progress afterwards. As a first step, we should be able to stop the game if the grid is full. Add a method `public boolean gameOver()` in `Grid` which returns `true` if the game is over.
  9. We must now take care of the winning conditions. We add a function `public boolean hasWon(int color)` which returns `true` if the player with the color `color` has won. As a first step, this function returns `true` only if the player has 4 disc aligned in a column. Also, you should add a `private boolean gameOver;` attribute which is returned by `gameOver` and set to `true` whenever someone has won (or the grid is full).
  10. Add the winning condition for the row.
  11. **Harder.** Add the winning condition for the ascending diagonals.
  12. **Harder.** Add the winning condition for the descending diagonals.
4. **Pro.** Is it easy to change the size of the grid in your code? And the number of discs to win? If not, then it should.

The next question is annotated “Plus Ultra”, that means it is an open question beyond the scope of the lab.

5. **Plus Ultra.** Extend the game in any direction you want. Use a new class `Connect4PlusUltra.java` so you can still enjoy the automated correction tool for the previous exercise. Be sure to describe your changes in the explanatory video you will make.

## Exercise 4 – Competitive track

Those interested in the competitive track must register here (you can join anytime): <https://docs.google.com/spreadsheets/d/1KMZx58SoE08g-14usphtaLFnPhKzBDhTpa9PgixOok8/edit?usp=sharing>.

We rely on the website <https://onlinejudge.org>, therefore you should create an account on it and include your UVa username in the source code of the problem. You can try your solution to the problem by clicking the “submit” button in the top right corner above the PDF. Place the challenges in the folder `src/`. In order to be compatible with the online judge, please follows these specifications<sup>1</sup>: [https://onlinejudge.org/index.php?id=15&Itemid=30&option=com\\_content&task=view](https://onlinejudge.org/index.php?id=15&Itemid=30&option=com_content&task=view). We propose three challenges for this time:

1. *Cost cutting*: [https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=24&page=show\\_problem&problem=2827](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2827)
2. *Divison of Nlogonia*: [https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=24&page=show\\_problem&problem=2493](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2493)
3. *The snail*: [https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=24&page=show\\_problem&problem=514](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=514)

In order to solve these problems, some additional information on `Scanner`:

- `scanner.nextInt()` ignores the spaces and newlines before the integers, so if you type “ 12”, there is no problem, and “12” will be read.
- You can read consecutive numbers by calling several times the method `scanner.nextInt()`.
- In order to test more easily your program, place the problem’s example in a file `input.txt`:

```
javac -d target src/Snail.java
java -cp target Snail < input.txt
```

The part `< input.txt` will send the content of the file `input.txt` on the standard input (*i.e.*, “in the scanner”) of the Java program. Therefore you don’t need to type it manually each time you want to test your program.

---

<sup>1</sup>Thanks to Léo for pointing it out!

