Lamis Armoush
Software Development Internship Assessment

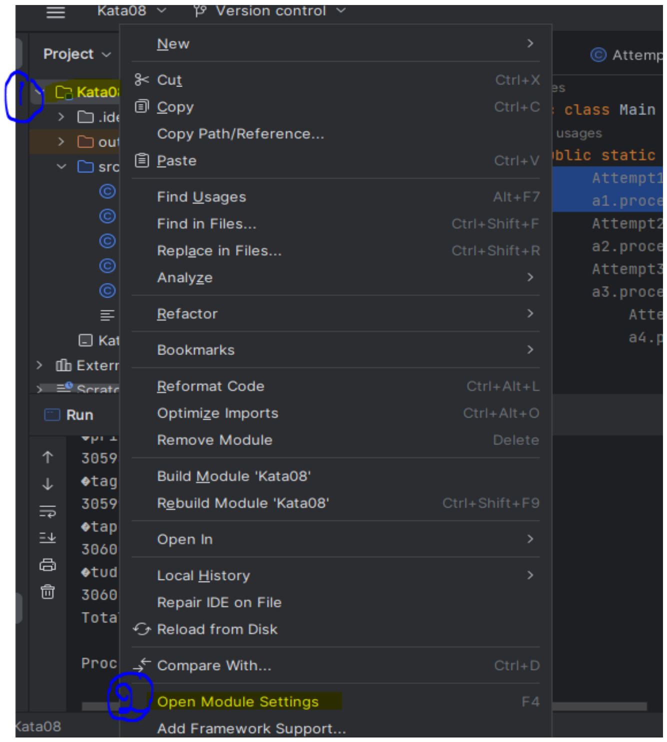
Setting up the system

1- Install IntelliJ IDE

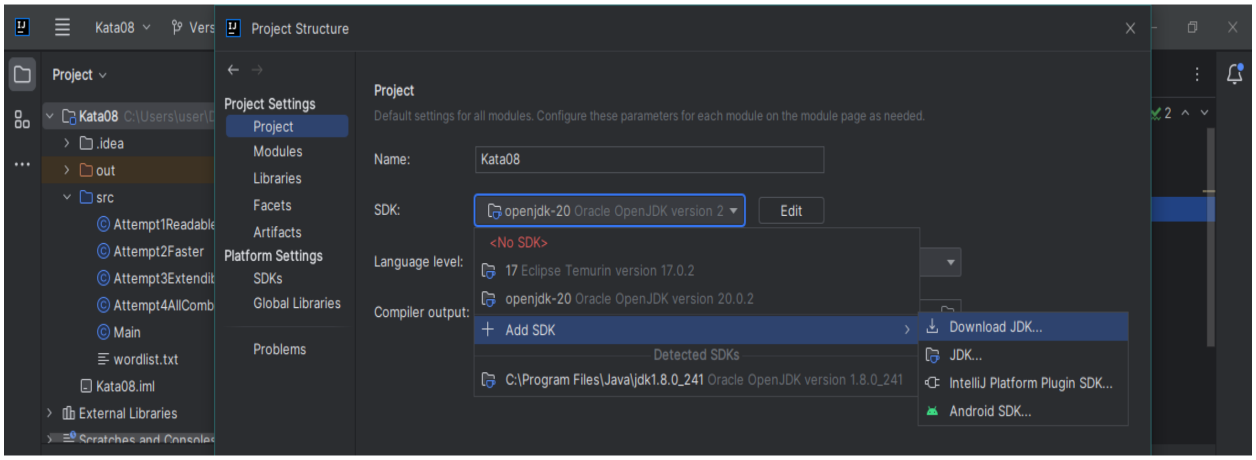
- Go to IntelliJ download page: https://www.jetbrains.com/idea/download/?section=windows
- Scroll down and click on the community edition download button.
- Choose the best version that suites your computer's operating system (windows, linux, mac).
- After downloading open the installer and follow the instructions.

2- Opening the project and installing JDK

- After opening IntelliJ select open project and navigate to file location ../Kudwa_Assessment_LamisArmoush/Kata04 and click open
- Now right click on project at the top of the screen and got to Open Module settings.

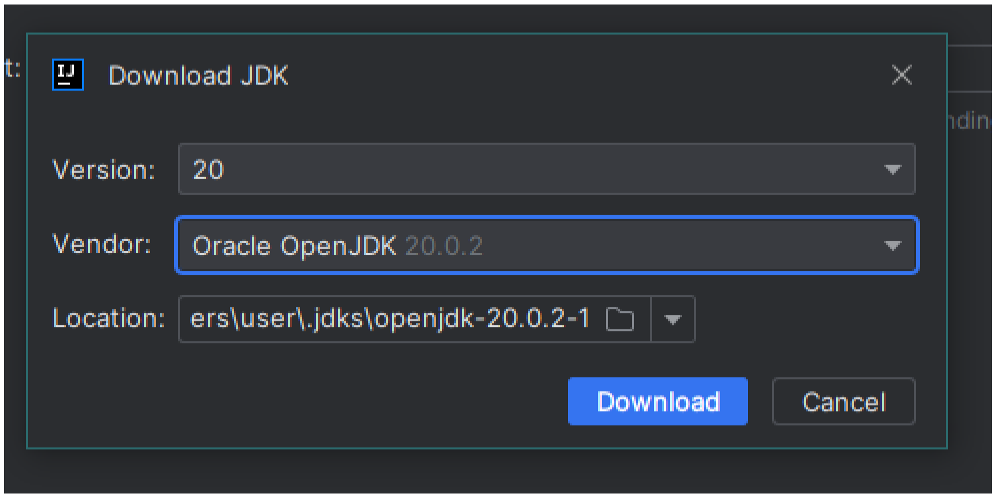


- A new window will open. Go to project>>SDK>>Add SDK >>download SDK



Project
Default settings for all modules. Configure these parameters for each module on the module page as needed.
Name: Kata08
SDK: openjdk-20 Oracle OpenJDK version 2
Language level:
Compiler output:

- Click on oracle OpenJDK 20.0.2 and click download.



Version: 20

Vendor: Oracle OpenJDK 20.0.2

Location: ers\user\.jdks\openjdk-20.0.2-1

- Now, you can click on src>>Main.java and run the file.

Lamis Armoush
Software Development Internship Assessment

Kata 04

Part One

The first part of this problem can be viewed in the class entitled PartOne:

In this class, we have 2 attributes “day_nb” that aims to hold the day_nb having the least spread, and “smallestSpread” storing that value.

We also have 3 methods:

- **readFile**: takes the path to the file as input and uses FileReader and bufferedReader to read the file and pass each line of values over to the method findSpread(). This method also aims to handle exceptions that are generated of a wrong path or fail to read the file.
- **FindSpread:** takes in the line of value and splits it into an array of values, then indexes the max and min temperature along with the day number, it also calculates the spread and compares it to the smallest spread storing it accordingly. Here, we also handle some exceptions like indexOut of boundException referring to an empty or dashed line, and NumberFormatException that refers to the numbers having a * or non-integer values. Within that block a code can be added later to handle these specific cases but as no specified instructions were indicated, we left this part for now.
- **smallest_spread** : this method calls in all other methods in order and prints out the result + returns a string indicating the day_nb.

Part Two

Part 2 was almost the same replica of part 1 with a minor change representing the indices of the For score, Against score, and teamName values. It can be viewed in the file entitled PartTwo.java

Part Three

To remove this duplication, we copied all the code into a new class called **CommonFunc** with a very minor change creating the attributes “max_index”, “min_index” and “id_index” and replacing the indices of the arrays below with those values. We also added the constructer that allocates these variables to a certain value. Now, both classes **PartThreeOne** and **PartThreeTwo** will inherit this **CommonFunc** and only have the constructor that sets these variables to the needed indices.

Instead of having the 2 classes inheriting this common function, we could have easily set those values by taking input and setting them in the common function’s constructor but splitting them comes in handy if we want to later on add other functions unique to each of those 2 classes.

Lamis Armoush
Software Development Internship Assessment

Kata-08

Part One: As Readable as Possible

This part can be viewed in the part entitled **Attempt1Readable.java .**

The approach followed here is very straightforward:

- we defined 2 lists wordsLessThan6 and wordsEqual6
- in the method **scanDictionary** we scan all the words and add those less than 6 letters to wordsLessThan6 and those having 6 letters to wordsEqual6
- now in the method **processDictionary** we scan by calling the previous method then loop over the wordsEqual6 taking in each word and splitting it in 2 different subs at different points. And at each point we check if the wordsLessThan6 list contains these subs. If yes, we print them along with the 6-letter word.

The time taken by this method is 21.059 seconds on average for an output of 30601 combinations resulting in a 6-letter word.

Part Two: As Fast as Possible

This part can be viewed in the part entitled **Attempt2Faster.java .**

At this level, we noticed that the main issue with the previous code's speed is the need to search with each and every iteration of 6 letter word among a huge list of words with length<6 and if we want to make it for larger words of 10 15 or even 20 letters it will get slower and slower with the list getting bigger. Moreover, as the list gets bigger, inserting takes more and more time. So, why not split these words into smaller closely related shunks.

The list storing wordsLessThan6 now becomes a dictionary with keys as first letter and value is another dictionary with keys equal length of each word. Now, list containing words like "all" and "app" can be referenced by calling key 'a' followed by key '3'.

With this approach, the time to store and find the combinations went down to an average of 0.764 seconds for the 30601 combinations.

Part Three: As Extendible as Possible

Extending the application might refer to a lot of options like adding an option to choose the number of letters for the concatenated word, or finding all possible combinations for all words, or among a certain range of word sizes.

The choice we made is present in Attempt3 that makes the user chose the number of letters of the concatenated word, and in Attempt4 that finds all combinations among a whole range of sizes.

In fact, it was simple to extend from the faster version as all we had to do was substitute the value 6 with a giving value assigned when calling the constructor. Then, from this level to Attempt4, we had to use the composition relation as Attempt 4 is composed of attempt3 and has 1 method: **process()** that loops over the range and calls Attempt 3’s methods to compute the combinations according to each value in this range.

Overall, time taken varies here according to the range with a max time of 17.562 seconds to find all possible combinations.