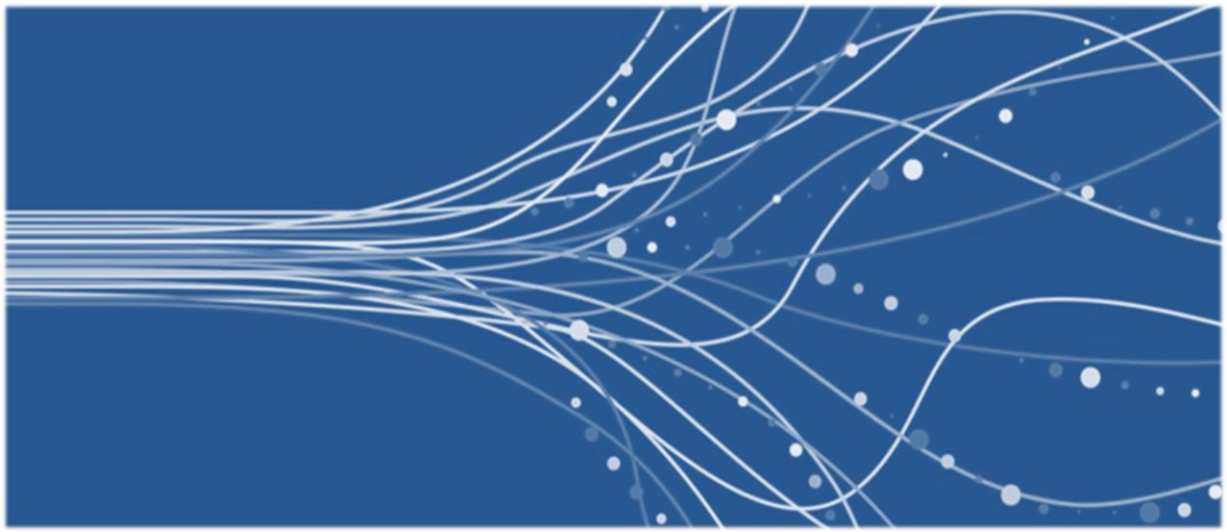


Multithreading Assignment

13.1.2023

Ohad Wolfman, ID 316552496

Zev Kehat, ID 203283908

Overview

This project was built as an assignment in an Object-Oriented Programming course, taken as part of our BSc studies at Ariel University.

The objective of the assignment was to understand and implement a number of different multithreading methods, while comparing and analyzing the results.

The assignment consisted of two parts:

1. Part 1 – Multithreading Comparison Program:
    a. Writing a program that would at first create a specified number of files, and randomly choose the number of lines in each file.
    b. After creating the files, we were instructed to write 3 different methods for counting the total number of lines in the files:
        1) Using just the main thread (standard function).
        2) Using multiple threads, created by a "Thread" extending class.
        3) Using a thread pool, created by a "Callable" implementing class.
2. Part 2 – Customized Multithreading Objects:
    a. Creating a generic "Task" class, that return results and may throw an exception.
    b. Creating a "CustomExecutor" class, used for executing the Task type tasks, according to the priority of the task.
    c. Besides the classes we were instructed to create, we built two extra classes, used to simplify, and correctly manage the two types:
        1) MyThreadPoolExecutor – a customized extension of the Java ThreadPoolExecutor.
        2) CustomFutureTask – a customized extension of the FutureTask, which implements the interface.
3. The listed classes, the test class and the project Readme were all handed in.

Code Overview and Methods

1. General Comments:

Programming Decisions – The assignment allowed us to make a few implementation decisions, which we decided as follows:

After at first writing the CustomExecutor as an extension of the ThreadPoolExecutor, we decided to create another class called MthreadPoolExecutor that would allow us more flexibility with the methods we created in the CustomExecutor.

Running the program – Part 1:

1) The main method is written in a way that will first find the user's current directory.
2) Call the constructor of Ex2_1 that get String currentPath will save the path to the files.
3) The first two methods are simple and static, so will have to be called from the class Ex2_1.
4) The first method is – createTextFiles that get (int n, int seed, int bound) and choose a number randomly from 1 to bound, in a random state seed.
   After that, the method generates files that contain some non-relevant rows.
   The other method will use those files, so we will be able to analyze runtime efficiency.
5) The second method – getNumOfLines - get String[] fileNames and counts the total number of rows in all the files by counting theme one by one, without using Threads.
6) The third method - getNumOfLinesThreads - get String[] fileNames and counts the total number of rows in all the files using Thread.
7) The fourth method – getNumOfLinesThreadPool - get String[] fileNames and counts the total number of rows in all the files using Thread.
8) At the end of running the main program, the output will list the number of lines in the files and the running time it took to run.
   For example for the bellow input, the program will print:

```
Ex2_1.createTextFiles(10, 2, 1000000);
```

The Program will print:

```
[file_1, file_2, file_3, file_4, file_5, file_6, file_7, file_8, file_9, file_10]
1: 4937566
it took 315 milliseconds
2: 4937566
the second took 248 milliseconds
3: 4937566
the third took 250 milliseconds
```

b. Running the program – Part 2:

1) The project's main method is added to the CustomExecutor class for convenience, but can be added in other classes as pleased.
2) Once a main method is added, create a CustomExecutor object. The executor will initialize with predefined parameters:
   `CustomExecutor executor = new CustomExecutor();`
3) After creating the executor, you may create new tasks for the executor to execute (run), in one of three createTask methods:
   a) Create a Task instance. Task is a generic class, so the user can choose which object type to use when calling the Tasks instance.
      `Task<V> task = Task.createTask(()->{…}, TaskType.<type>};`
   b) Create a Callable object and submit it to the executor.
      `Callable<Double> callable1 = ()-> {…)};`
   c) Create a Callable object and a TaskType object, and submit them to the executor.
      `customExecutor.submit(()->{…}, TaskType.COMPUTATIONAL);`
4) When submitting a new task using one of the methods above, save the returned result as a Future object with a variable name, in order to access the tasks results at a later stage.
5) When finishing the use of the executor, be sure to shut it down with the "teminateGracefully" method.

2. Task.java:

a. A generic type, which implements the interfaces "Callable" and "Comprable".
b. Is made up of two objects – A callable called "task" and a newly created type called '"tasktype".
c. The class contains a private constructor, triggered by two constructors available to users.
d. The class has getter/setter methods for the task's task, and the task's priority.

e. The class has a "call()" method, used when executing a callable thread.

f. The class is comparable, implementing it own compareTo method.

3. CustomFutureTask.java:

   a. A generic type, which is an extension of the "FutureTask" class, and implements the "Comprable" interface.

   b. When a Task is called to be executed, the task's call method is wrapped and handled by an instance of a CustomFutureTask, allowing certain access to the task after wrapped already has a runnable.

   c. The class contains a constructor, a getTask method, and a compareTo method.

4. MyThreadPoolExecutor.java:

   a. An executor type, which is an extension of the ThreadPoolExecutor class.

   b. The class performs the actual transformation of "Task<V<" types to RunnableFutures, using the "submit" and "newTaskFor" methods.

   c. The executor also contains a constantly updated tracker of the Max Priority value (lowest priority) in the current task queue. This value is retrieved using the "getMaxPriority" method.

   d. The executor will terminate when a "gracefullyTerminate" is called, but first wait for the termination of running threads before doing so.

5. CustomExecutor.java:

   a. An executor object manager, which holds a "MyThreadPoolExecutor" type, and receives and executes tasks from the user.

   b. The CustomExecutor can get a task either as a Task object, a "callable" object, or a pair of a callable and a TaskType objects.

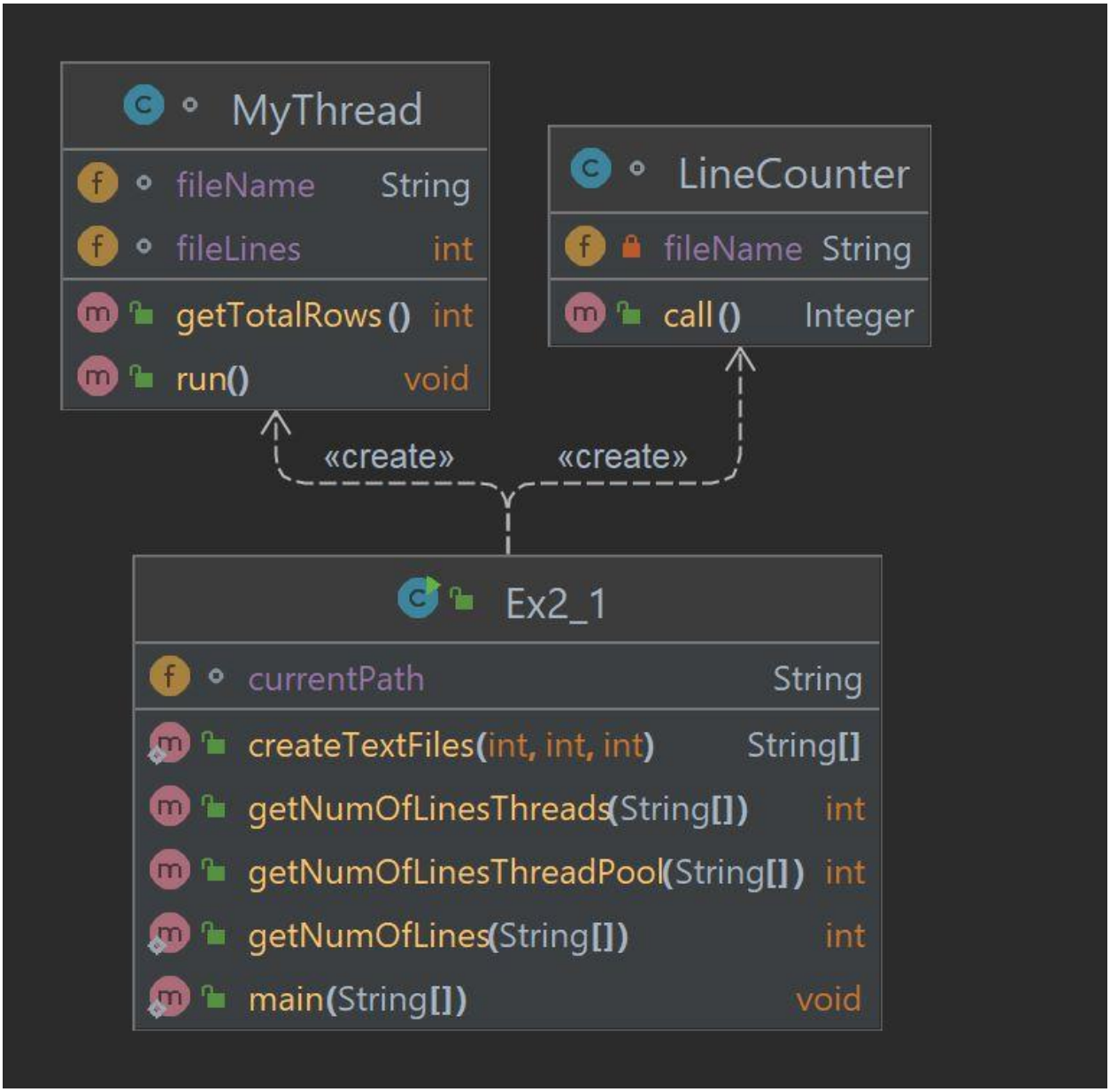
   c. Any task accepted will be submitted to the executor as a task (after a conversion to one, if needed) for execution.

   d. The CustomExecutor can retrieve the current MaxPriority waiting in the Thread Pool Queue.

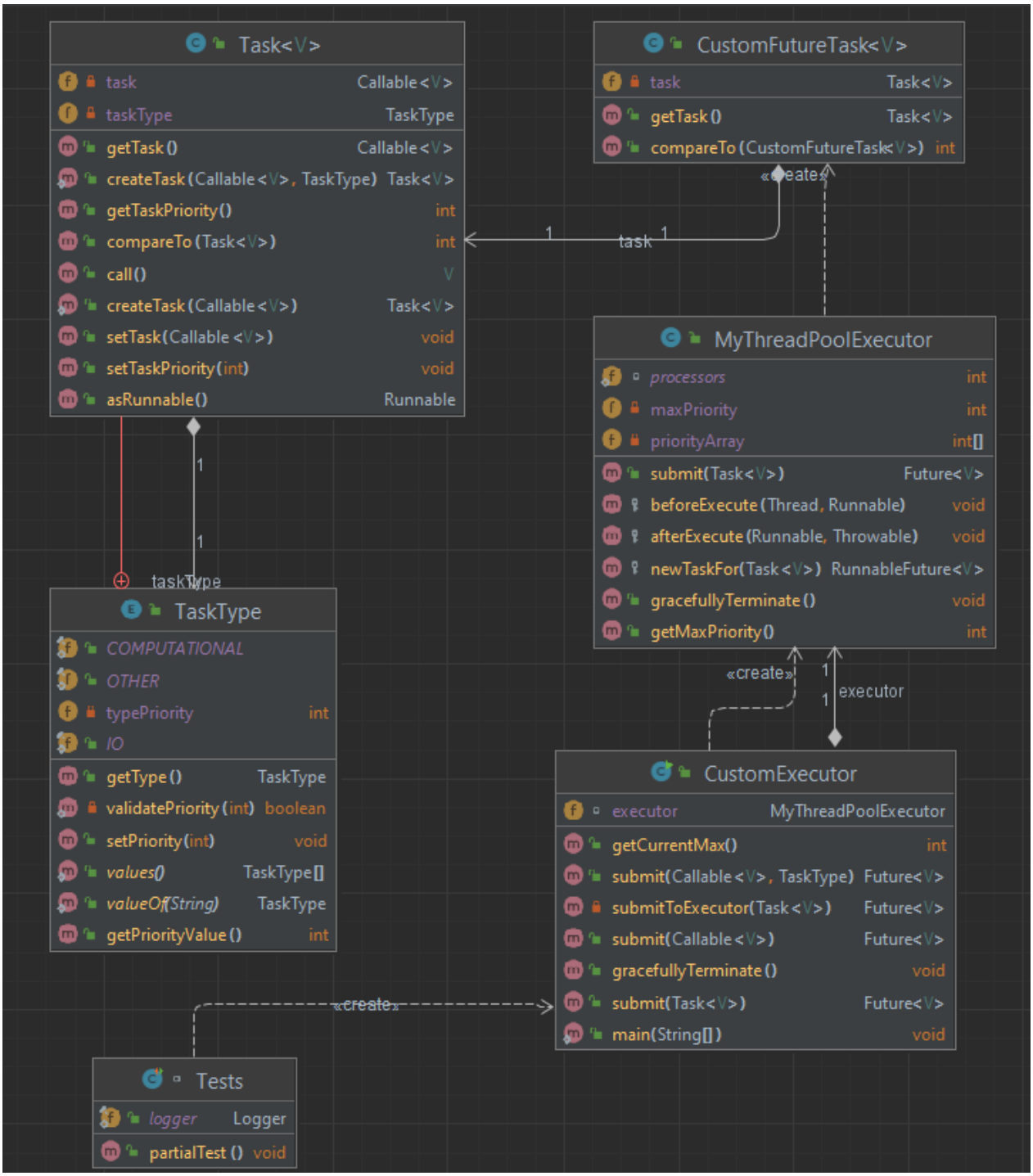
   e. When finished using the CustomExecutor, it must be closed. This is done by calling the "gracefullyTerminate" method.

Project Diagram

Ex2_1 – part one:



Ex2_2 – part two:



Summary

The assignment presented a number of difficult challenges, in planning and executing the design pattern of the project. The assignment demanded that we deeply research different Types of objects, and how they can assist us in a successful implementation, such as choosing the right executor service to be used, understanding the difference between future types, and how each one can come in handy, understanding the process of the queue, and therefore how we can add objects in a prioritized way, and more.

After viewing the results of the different tests run, both in Part 1 and in Part 2 of the project, two things are clear to us:

1. The regular way to count the rows is the worst efficiency, since it does not perform tasks at the same time and it is required to go file by file and count the lines in it.
2. Multithreading has great advantages – the "parallel" execution of tasks proved it's efficiency and beat the single threads in every trial run.
3. The thread pool requires a lot of resources in creating the threads, so a lot of time running the program is wasted since the operation of calculating the rows is a very easy operation and does not require a lot of computing power.
4. In general - MultiThreads are complex! Using a few multiple threads for complex tasks can become very tricky, and a deep understanding of the processes and dependencies is crucial.