SPL211 - ASSIGNMENT 2

Java Generics, Concurrency, and Synchronization
TAs in charge:
Sohel Kanaan
Yair Vaknin

Publication date: **22.11.2020**
Deadline: **29.11.2020 23:59**
Unit tests submission deadline: **13.12.2020 23:59**

Before You Start

- The goal of the assignment is to practice concurrent programming on Java 8 environment. This assignment requires a good understanding of Java Threads, Java Synchronization, Lambdas, and Callbacks. Make sure you revise the lectures and practical sessions that cover these topics.
- **While you are free to develop your project in an environment of your choice, the projects will be tested and graded on a CS LAB UNIX machine. Therefore, it is recommended that you run your assignment on a lab unix machine before submission.**
- The Q&A of this assignment will take place at the course forum only. Critical updates about the assignment will be published in the assignment page on the course website. These updates are mandatory, and it is your responsibility to be updated. A number of guidelines for using the forum:
    - Read previous Q&A carefully before asking a new question. Repeated questions might be left unanswered.
    - You are NOT allowed to post any kind of solution and/or source code in the forum as a hint for other students. In case you feel that you have to discuss any such matters, please use the staff reception hours.
- Dolav is the only staff member who can approve extensions. In an appeal for an extension, please contact him directly.

Good Luck!

1 GENERAL GUIDELINES

- Read the javadocs of all the interfaces we provide you with.
    - Please stick to the java documentation of the classes and methods. Do not add public members/methods to the supplied classes (unless else specified in this document). Do not change the signature of the supplied methods.
    - You can add the word “**synchronized**” to the signature of any method.
    - You can throw exceptions from any method.
- Make your code as concurrent and as efficient as possible.
    - Java introduces a collection of data structures that are safe under concurrent access. Although it might be very convenient to use those data structures - do so only if there’s no better (simple) solution.
    - The least you synchronize your code - the better. You still need to use synchronization where it is inevitable. Think Twice.
    - The efficiency of your implementation will affect your grade. However, efficiency must not come at the cost of correctness.

2 INTRODUCTION

In the following assignment you are required to implement a simple Microservice framework, which you will later use to implement a system for the Jedi Order (more on that in the next section. For now, please just focus on the framework).
In the Microservices architecture, complex applications are composed of small and independent services that are able to communicate with each other using messages. The Microservice architecture allows us to compose a large program from a collection of smaller independent parts.

This assignment is composed of two main sections:

1. Building a simple Microservice framework.
2. Implementing a system on top of this framework.

<u>It is very important to read and understand the entire work prior to implementation. Do not be lazy here.</u>

3 PRELIMINARY

In this section you will implement a basic Future<T> class, which you will use later in the assignment. A Future<T> object represents a promised result - an object that will eventually be resolved. The class allows retrieving the result once it is available. Future<T> has the following methods:

- T get(): retrieves the result of the operation. This method waits for the computation to complete in the case that it has not yet been completed.
- resolve(T result): called upon the completion of the computation, this method sets the result of the operation to a new value.
- isDone(): returns ‘true’ if this object has been resolved, and ‘false’ otherwise.
- T get(long timeout, TimeUnit unit): retrieves the result of the operation if available. If not, waits for at most the given time unit for the computation to complete, and then retrieves its result (if resolved).

4 PART 1: SYNCHRONOUS MICROSERVICES FRAMEWORK

4.1 DESCRIPTION

In this section you will build a simple Microservice framework. Such a framework consists of two main parts: a Message-Bus, and Microservices. Each Microservice is a thread that can exchange messages with other Microservices, using a shared object - the Message-Bus. There are two different types of messages:

1. **Event**:

An Event defines an action that needs to be processed. Each Microservice specializes in processing one or more types of events. Upon receiving an event, the Message-Bus assigns it to the messages queue of a certain Microservice - one that registered to handle events of this type. It is possible that there are several Microservices that can handle the event that was just received. In that case, the Message-Bus assigns the event to one of them, in a round-robin manner (described below).

Each Event expects a result of type Future<T>. This Future<T> object should be resolved once the Microservice that handles the Event completes processing it. For example: In the scenario you will run on the assignment, R2D2 needs to update its friends when the mission it is assigned to (deactivating the shield generator) is accomplished. R2D2 cannot send messages, though. Therefore, it will update its friends via resolving a Future<T> object.

2. **Broadcast**:

Broadcast messages represent a global announcement in the system. Each Microservice can register to the type of broadcast messages it is interested in. The Message-Bus sends the broadcast messages that are passed to it to all the registered Microservices (unlike Events - those are sent to only one of the registered Microservices).

4.2 ROUND ROBIN

As specified, when several Microservices are registered to handle the same type of Events, those Events are sent to them in a round-robin manner. It means that the Microservices “take turns” in handling those Events. For example, if there are three AttackEvents that need to be handled by the two Microservices named HanSolo, C3PO that can handle them, then e.g. AttackEvent #1 will be sent to HanSolo, AttackEvent #2 will be sent to C3PO, and AttackEvent #3 will be sent to HanSolo.

4.3 EXAMPLE

Now we’ll partly describe the flow of the whole system. It may help you understand how the Microservices framework operates.

On our example there are four Microservices:

- Leia: she sends AttackEvents that are handled by HanSolo and C3PO.
- HanSolo, C3PO: handle AttackEvents.
- R2D2: handle DeactivationEvent.

In addition, there are two types of messages:

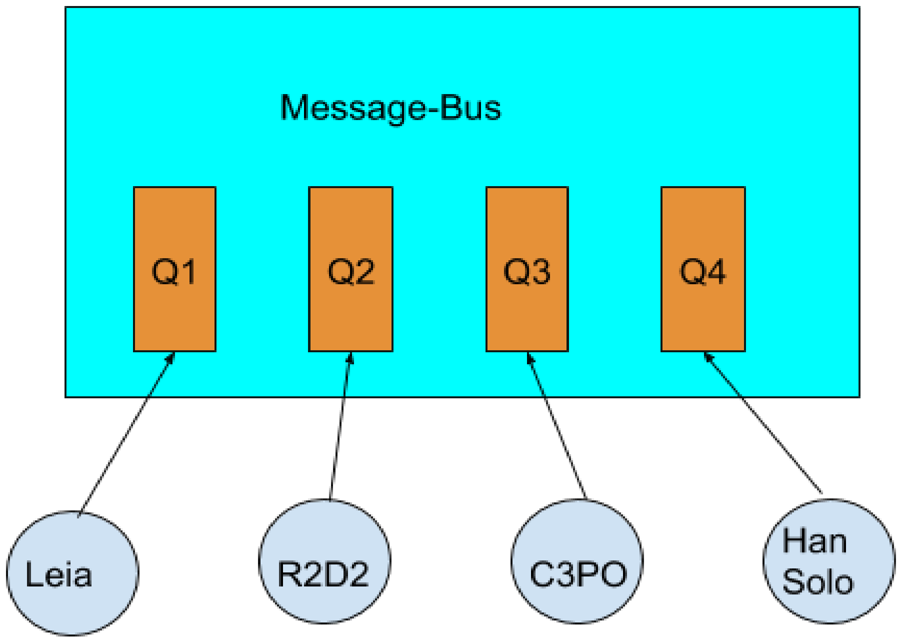
- AttackEvent
- DeactivationEvent

4.3.1 INITIALIZING A SUBSCRIBER

- Each Microservice should register itself to the Message-Bus and initialize itself. On initialization, each Microservice subscribes to the messages that it is interested to receive. In our example HanSolo and C3PO will handle AttackEvents, and R2D2 will handle DeactivationEvent.

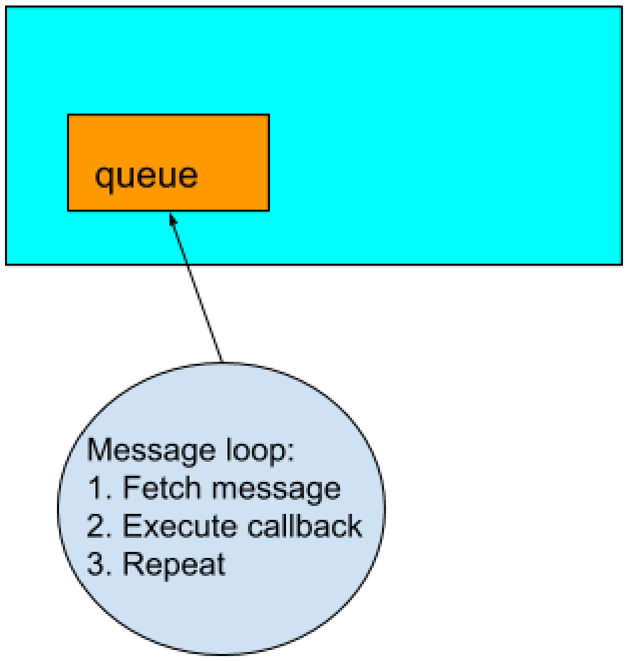
- A Microservice that is registered to handle a certain type of Messages, has a callback function that defines its functioning upon receiving such a Message. This callback function will be called by Microservice to process the Message.
-

Figure 1 describes the initialization phase:



4.3.2 MESSAGE LOOP PATTERN

In this part you will implement the message-loop design pattern. In such a pattern, each Microservice is a thread that runs a loop. In each iteration of the loop, the thread tries to fetch a message from its queue and process it. Figure 2 describes the message-loop in our system:

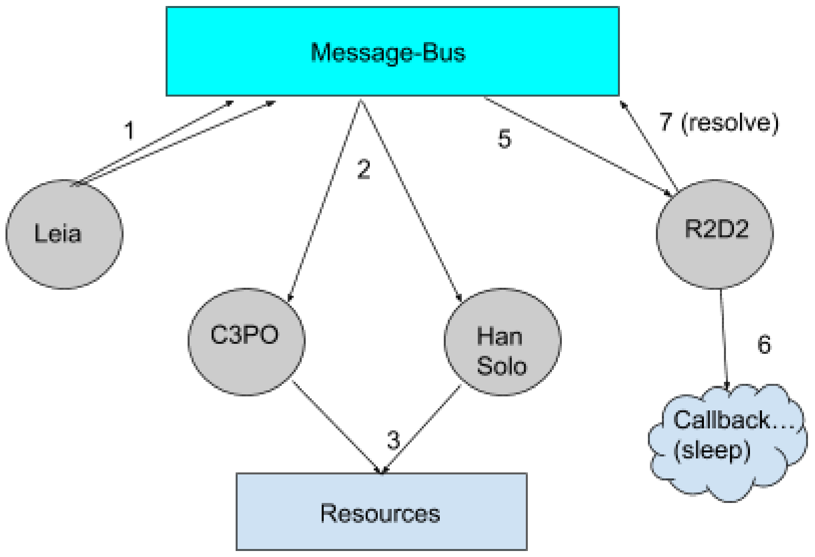


4.3.3 PROCESSING EVENTS

Figure 3 describes the interaction between the Microservices and the Message-Bus:

1. Leia sends two AttackEvents to the Message-Bus. The Message-Bus sends them in a round robin manner, and inserts the first to the queue of HanSolo, and the second to the queue of C3PO.
2. HanSolo and C3PO fetch the events, and the corresponding callback is called.
3. In the callback, there is a need to acquire resources. HanSolo and C3PO ask for resources simultaneously (beware deadlocks, take care for proper synchronization if needed). After the resources for each attack are acquired by a thread, the thread simulates the attack by “sleeping” for a given amount of time.
4. When HanSolo and C3PO finish their missions, someone needs to inform R2D2 that it can now deactivate the shield generator. A DeactivationEvent is sent (by who? This is up to your implementation. We did not elaborate on that on the diagram).
5. R2D2 fetches the DeactivationEvent.
6. R2D2 calls the corresponding callback. In our program, the deactivation of the shield generator is simulated by a sleep of the thread for a given amount of time.
7. Now that the R2D2 deactivated the shield generator, it updates his (“its”, actually) friends.

IMPORTANT - this is only a partial flow. The complete mechanism is richer in details and requires more types of Messages. It will be fully described later on.



4.4 IMPLEMENTATION INSTRUCTIONS

When building a framework, one should change the way they think. Instead of thinking like a programmer who writes software for end users, one should now think like a programmer writing software for other programmers. Those other programmers will use this framework in order to build their own applications. In this part of the assignment you will build a framework, and the programmer who will use your code in order to develop their application is the future you when eventually you'll work on the second part of the assignment. Attached to this assignment is a set of interfaces that define the framework you are requested to implement. The interfaces are located at the bgu.spl.mics package. Read the javadoc of each interface carefully. You are only required to implement the Message-Bus & Microservice classes at this part. The following is a summary and additional clarifications about the different parts of the framework.

- Message: a data-object which is passed between Microservices (via the Message-Bus only!) as a means of communication. The Message interface is a Marker interface. That is, it is used only to mark other types of objects as messages.
- Broadcast: a Marker interface extending Message. When sending Broadcast messages using the Message-Bus, it will be sent to all the Microservices that are registered to receive this Broadcast type.
- Event: a marker interface extending Message. A Microservice that sends an event message may expect to be notified when the Microservice that processes the event has

completed processing it. The event has a generic type variable T, which indicates its expected result type. The Microservice that has received the event will call the method ‘Complete’ of the Message-Bus once it has completed treating the event, in order to resolve the result of the event.

- Microservice: an abstract class that each Microservice in the system must extend. The abstract Microservice class is responsible to get and manipulate the singleton Message-Bus instance. Derived classes of Microservice should never directly deal with the Message-Bus. Instead, they have a set of internal protected wrapping methods they can use. When subscribing to a message type, the derived class also supplies a callback function. The Microservice stores this callback function together with the type of the message it is related to. The Microservice is a Runnable (i.e. can be executed as a thread) - its run method implements a message loop. When a new message is taken from the queue, the Microservice invokes the appropriate callback function.
  When the Microservice executes the run method, it registers itself to the Message-Bus, and then calls the abstract ‘initialize’ method. This method allows derived classes to perform any required initialization code (e.g. subscribing to messages). Once the initialization code is completed, the actual message-loop starts. The Microservice can fetch messages from its messages queue using the Message-Bus’s ‘awaitMessage’ method. For each message it should execute the corresponding callback. The Microservice class also contains a terminate method that should signal the message-loop that it should end.

  IMPORTANT: Registration, initialization, and unregistration of the Microservice must be executed inside its run method.

- Message-Bus: A shared object used for communication between Microservices. It should be implemented as a **thread-safe singleton (!!)**. The Message-Bus interface will be implemented within the MessageBrokerImpl class (provided to you). There are several ways to implement the Message-Bus methods. Be creative and find a good, correct and efficient solution. **Notice, fully synchronizing this class will affect all the performance of the system (and your grade) - try to find ways to avoid blocking threads as much as possible.**
  The Message-Bus creates a queue for each Microservice who calls the ‘register’ method. When a Microservice calls the ‘unregister’ method of the MessageBroker, the Message-Bus should remove its queue and clean all references related to that Microservice. Once the queue is created, a Microservice can fetch messages from its queue using the ‘awaitMessage’ method. The ‘awaitMessage’ method is blocking **(!!)**, that is, if there are no messages available in the Microservice queue, the calling thread will wait until some message is available.

  Message-Bus methods elaborated:
  - register: a Microservice calls this method to register itself. This method should create a queue for the Microservice in the Message-Bus.

- subscribeEvent: A Microservice calls this method to subscribe itself for some type of event (the specific class type of the event is passed as a parameter).
- subscribeBroadcast: A Microservice calls this method to subscribe itself for some type of broadcast message (The specific class type of the broadcast is passed as a parameter).
- sendBroadcast: A Microservice calls this method to add a broadcast message to the queues of all the Microservices that are interested in it.
- Future sendEvent(Event e): A Microservice calls this method to add the event e to the messages queue of one of the Microservices that are subscribed to receive events of type e.getClass(). The messages are added in a round-robin manner. This method returns a Future object - from which the sending Microservice can retrieve the result of processing the event once it is completed.
- void complete(Event e, T result): A Microservice calls this method to notify the Message-Bus that the event was handled. The Microservice provides the result of handling the request. The Future object associated with event e should be resolved to the result given as a parameter.
- unregister: A Microservice calls this method in order to unregister itself from the Message-Bus. The Message-Bus will remove the messages queue allocated to the Microservice, and clean all the references related to it.
- awaitMessage(Microservice m): A Microservice calls this method to fetch a message from its queue. This method is blocking **(!!)**.

5. PART 2 - THE FULL SYSTEM

5.1 BACKGROUND

A long time ago in a galaxy far far away… Luke Skywalker has returned to his home planet of Tatooine in an attempt to rescue his friend Han Solo from the clutches of the vile gangster Jabba the Hutt. Little does Luke know that the GALACTIC EMPIRE has secretly begun construction of a new armored space station even more powerful than the first dreaded Death Star. When completed, this ultimate weapon will spell certain doom for the small band of rebels struggling to restore freedom to the galaxy...

5.2 OVERVIEW

In this part you will build a system with which the rebels, headed by Princess Leia, will coordinate their attack on the empire. In order to build this system, you will use the Microservice framework from part 1.

The end-to-end description of our system is as follows: Leia is initialized with Attack objects, which she sends to HanSolo and C3PO as AttackEvents. In order to process those events, HanSolo and C3PO use the Ewoks - the resources in our program (in reality, though, Ewoks are little forest creatures that can fight fairly well). Whenever HanSolo or C3PO hold the required resources for an attack - they execute it (this is simulated by “sleeping”). When all attacks are finished, R2D2 is informed and deactivates the shield generator (simulated by “sleeping”). Finally, Lando can now bomb the star destroyer of the Empire (simulated by “sleeping”). Only when this is done - all threads terminate at once **(!!)**.

5.3 PASSIVE OBJECTS

This section contains information about the required data classes (a.k.a. non-runnable classes). The name of the class appears first, then a short description. Those are followed by the mandatory members and methods, and additional instructions about the members and methods you can add to the class if you see right.

- Ewok - a forest creature.
    - Member: int serialNumber.
    - Member: boolean available. Indicates whether this resource is free, or held by some thread.
    - Method: acquire(). Change the ‘available’ flag from ‘true’ to ‘false’.
    - Method: release(). Change the ‘available’ flag from ‘false’ to ‘true’.
    - You can add to this class members and methods as you see right.

- Ewoks - wraps a collection of Ewok objects.s
    - Holds a collection - of your choice - of Ewoks objects.
    - All calls for resources (by HanSolo and C3PO) are done via the Ewoks class.
    - You can add to this class members and methods as you see right.

- Attack - information about an attack.
    - List<Integer> serialNumbers. Indicates the required Ewok objects for the attack.
    - int duration - duration of the attack in milliseconds. The thread that executes the attack will simulate it by sleeping for that duration.
    - Leia is initialized with a list of Attack objects. She sends each of them as an AttackEvent to the Message-Bus.
    - Do not add any additional members/method to this class (except for a constructor, and getters).

- Diary - in which the flow of the battle is recorded. We are going to compare your recordings with the expected recordings, and make sure that your output makes sense.
    - int totalAttacks - the total number of attacks executed by HanSolo and C3PO. can also be of AtomicInteger type. Stamped only by HanSolo or C3PO **(!!)**.
    - long HanSoloFinish - a timestamp indicating when HanSolo finished the execution of all his attacks.

- long C3POFinish - a timestamp indicating when C3PO finished the execution of all his attacks.
- long R2D2Deactivate - a timestamp indicating when R2D2 finished deactivation the shield generator.
- long LeiaTerminate - a time stamp that Leia puts in right before termination.
- long HanSoloTerminate - a time stamp that HanSolo puts in right before termination.
- long C3POTerminate - a time stamp that C3PO puts in right before termination.
- long R2D2Terminate - a time stamp that R2d2 puts in right before termination.
- long LandoTerminate - a time stamp that Lando puts in right before termination.
- To get those timestamps, simply use System.currentTimeMillis().
- We will check that your timestamps make sense.
- Each timestamp is recorded by the specified name, e.g. only C3PO is allowed to set the value of C3POFinish. The totalAttacks member is recorded **only by HanSolo or C3PO**.
- You can add to this class members and methods as you see right.

5.4 MESSAGES

The following message classes are mandatory. The fields that these messages hold are omitted. Apply your reasoning and complete what's needed.

- **AttackEvent**
- **DeactivationEvent**
- **BombDestroyerEvent** - sent to Lando when all is set for his final action.

You may create more types of messages to get things done, if you find it necessary.

5.5 ACTIVE OBJECTS (Microservices)

This section contains the description of the required Microservices in our program (i.e. classes that extend the Microservice abstract class). We supplied you with code in which every Microservice has a single constructor. **Do not add another constructor!**
Other than constructors, you can add members and methods to the Microservices as you see right.
Remember: Microservices MUST NOT know of each other. The Microservices communicate indirectly, via the Message-Bus.

- **Leia**

Princess of the planet Alderaan, a member of the Imperial Senate and an agent of the Rebel Alliance. Initialized with Attack objects, and sends them as AttackEvents

- **HanSolo**

A pilot. Receives AttackEvents, and uses the Ewoks (those resources are also accessed by C3PO, Please pay attention to that) to execute them.

- **C3PO**

A droid programmed for etiquette and protocol. Receives AttackEvents, and uses the Ewoks (those resources are also accessed by HanSolo, please pay attention to that) to execute them.

- **R2D2**

A droid. After Han and C3PO finish their coordinated attacks on the Empire's troops, he ("it", actually) can deactivate the shield generator - so Lando can safely approach the star destroyer. The deactivation of the shield generator is simulated by sleeping.

- **Lando**

Lando Clarissian - an old friend of HanSolo, and a decent pilot. When informed about the deactivation of the shield generator (via BombDestroyerEvent) - he proceeds and bombs the star destroyer. This act is simulated by sleeping.

- Let us name another two heroes of the rebels, even though they don't take part in this assignment. **Luke Skywalker**, a Jedi master. Grew up as a farm boy, in Tatooine. And **Chewbacca**, a Wookiee and a co-pilot of Han Solo aboard the Millennium Falcon.

Important: All threads must terminate "at once"**(!!)**, i.e. the termination of all threads must be triggered by the same event. Also, make sure all threads terminate gracefully.

Tips for implementation:

1. Make sure you understand the flow of the battle. You can draw a diagram that describes the events exchange between the threads.
2. Use blocking methods carefully.
3. When resources are acquired, a deadlock may occur. How would you avoid it?
4. Understand the behavior of the collections you use and try always to find the best collection for your needs. For example: BlockingQueue has a blocking method – where do you need it? Where don't you need it?

5.6 INPUT FILES AND PARSING

5.6.1 JSON FORMAT

The input file for this assignment will be given as a JSON. You can read about JSON syntax here.
In Java, there are a number of different options for parsing JSON files. You can look at the 'Gson' library. See the Gson User Guide and APIs, there are a lot of informative examples there.

5.6.2 INPUT/OUTPUT FILES

The input file holds a JSON object which contains the following fields:

- Attacks: An array that represents the Attack objects. Each element in this array stands for a single attack, and holds the information about the duration of the attack and the requested Ewok objects. Leia is initialized with the Attack objects that are parsed from this part of the JSON.
- long R2D2: R2D2 is initialized with this parameter, which indicates the time it takes for him ("it", actually) to deactivate the shield generator.
- long Lando: Lando is initialized with this parameter, which indicates the time it takes for him to eliminate the star destroyer.
- int Ewoks: number of Ewok objects in our program. Each of the created Ewok objects will have a serial number. E.g. in case int Ewoks = 4, there will be 4 Ewok objects with the serial number 1,2,3,4.

Your program should output a JSON with all the fields and corresponding values that appear in the Diary.

Input/output example files are attached.

Your program will take two command line arguments - the first is the path to the input file, the second is the path (and the name) of the output file you should generate.

5.7 PROGRAM EXECUTION

The Main class will run the simulation. When started, it should accept as command line argument the names of the input/output files - the first argument will be the input file and the second is the output file.

The Main class will read the input file, and construct the passive objects and Microservices accordingly. Only after all threads terminate, the Main class will generate the output files and exit.

5.8 JUnit Tests

Testing is an important part of developing. It helps us to make sure our project behaves as expected. In this assignment, you will use Junit for testing your project. You are required to write unit tests for the following classes:

- Message-Bus
- Future
- Ewok

In order to help you understand the expected tests format, we included the class FutureTest. You can find there a test of the method Future.Resolve(). Complete the other required tests.

You need to submit the unit tests by 29.11.2020.

5.9 TESTING YOUR APPLICATION

- We supplied you with one input file which you can use as an example. Please test your application with more input files - of your own - and check different scenarios. We will test your program with several input files.
- What to check:
    - Correctness: the timestamps that your program generates must make sense.
    - Consistency: make sure that resources that are held by one thread, so not serve another thread.
    - Round-robin scheduling: make sure you implement it right.

5.10 BUILDING THE APPLICATION: MAVEN

In this assignment you will use maven as your build tool. You should read a little about how to use it properly. In order to help you, we supplied you with a maven pom.xml file (in the code attached to this assignment) that you can use to get started. You can learn about this file and how you can add dependencies to it.

5.11 SUBMISSION INSTRUCTIONS

5.10.1 FILE STRUCTURE

Attached to this assignment is a (partial) maven project you can use in order to start working on your assignment. In order for your implementation to be properly testable, you must conform to the package structure as it appears in the supplied project.

Included in the attached project -

- Interfaces and skeleton of the entire project.
- The files hierarchy, and the folder in which you should submit your tests.
- A POM file.

5.10.2 SUBMISSION

- Submit all the packages (including the unit tests).
- You need to submit the Unit Tests as well, therefore your POM should include a dependency for JUNIT.
- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff in order to submit without a pair. You cannot submit in a group larger than two.
- **Submit a '.tar.gz' file with all your code. The file should be named "assignment2.tar.gz". Note: We require you to use a '.tar.gz' file. Files such as '.rar', '.zip', '.bz', or anything else which is not a '.tar.gz' file will not be accepted and your grade will suffer.**
- **The submitted compressed file should contain the 'src' folder and the 'pom.xml' file only! No other files are needed. After we extract your compressed file, we should get one src folder and one pom file (not inside a folder).**
- Extension requests are to be sent to Dolav Nitay. Your request email must include:
  - Your name and your partner's name.
  - Your id and your partner's id.
  - Explanation regarding the reason for the extension request.
  - Official certification.
- **Requests without a compelling reason will not be accepted.**

5.12 GRADING

The assignments will be checked and graded on Computer Science Department Lab Computers - so be sure to test your program thoroughly on them before your final submission.
Grading will tackle the following points:

- Your application design and implementation.

- Tests will be run on your application. Your application must complete successfully and in reasonable time.
- Liveness and Deadlock: causes of deadlock, and where in your application deadlock might have occurred had you not found a solution to the problem.
- Synchronization: what is it, where have you used it, and a compelling reason behind your decisions. Points will be reduced for overusing synchronization and unnecessary interference with the liveness of the program.
- Checking if your implementation follows the guidelines detailed in “Documentation and Coding Style”.
- All threads should terminate gracefully**(!!)**.

Good Luck!