SPL201 - ASSIGNMENT 2

Java Generics, Concurrency, and Synchronization

TAs in charge:

Yair Vaknin

Elad Sulami

Publication date: **5.12.2019**
Deadline: **26.12.2019 23:59**
Unit tests submission deadline: **12.12.2019 23:59** (see section 5.8).

Before You Start

- The goal of the following assignment is to practice concurrent programming on the 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 which cover these topics.
- **While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine before submitting it.**
- 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 within your own responsibility to be updated. A number of guidelines for using the forum:
    - Read previous Q&A carefully before asking a new question; repeated questions will most probably go unanswered.
    - Be polite, remember that the course staff does this as a service for the students.
    - 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.
- Majeed is the only staff member who can authorize extensions. In case you require an extension, please contact him directly.

Good Luck!

1 GENERAL GUIDELINES

- Read the javadocs of all the interfaces we provided to you.
    - You must stick to the java documentation of each class and each method. For classes, you must add data members only with the allowed access levels. For methods, you must NOT change its return value type, parameters it receive, and the Exceptions types it throws.
    - You cannot throw exceptions which are not specified in the java documentation of the method. For example - if it is not stated in the documentation that a method throws an exception, then you must not throw exception (in this case, DO NOT add the keyword throws in the header of the method).
    - You can add the word “synchronized” to any method if needed.
- You should try to make your code as concurrent and as efficient as possible.
    - Java introduces a collection of concurrent data structures that can help you in writing a concurrent code. In this collection, the different data structures are implemented in such a way that different threads can use the data structure with as less synchronization and blocking as possible – (it is much more efficient than the naïve solution of synchronization methods of the data structure).
    - Read the Javadoc of each data structure you use – try to understand the level of concurrency each data structure allows (and if it is thread safe at all) and the different features of it (.e.g, blocking data structure).
    - Try to synchronize as less as possible – you still need to use synchronization where it is not avoidable.
    - The performance of your implementation can affect your grade. However, efficiency must not come at the cost of the correctness of the required implementation.

2 INTRODUCTION

In the following assignment you are required to implement a simple Pub-Sub framework, which you will later use to implement a system for the MI6, *On Her Majesty’s Secret Service* (that is the sixth in *James Bond* Series, what implies the properties of the system you are to implement. More on that in the next section. For now, please just focus on the Pub-Sub framework).
Pub-Sub is shorthand for Publish-Subscribe messaging, an asynchronous communication method in which messages are exchanged between applications without knowing the identity of the sender or recipient.

This assignment is composed of two main sections:

1. Building a simple Pub-Sub framework.
2. Implementing a system on top of this framework.

It is very important to read and understand the entire work prior to implementing it. Do not be lazy here.

3 PRELIMINARY

In this section you will implement a basic Future<T> class which you will use during the rest of the assignment. A Future<T> object represents a promised result T - 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.
- 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 PUB-SUB FRAMEWORK

4.1 DESCRIPTION

In this section you will build a simple Pub-Sub framework.
This framework is managed by a MessageBroker, there are 4 core concepts make up the Pub-Sub model:

1. Topic – An intermediary channel that maintains a list of subscribers to relay messages that are received from publishers.
2. Message – Serialized messages are sent to a topic by a publisher which has no knowledge of the subscribers.
3. Publisher – The application that publishes a message to a topic.
4. Subscriber – An application that registers itself with the desired topic in order to receive the appropriate messages.

The MessageBroker is incharge of:

1. Creating topics.
2. Registering subscribers to topics.
3. Delivering messages sent by publishers to the subscribers.

Note that an application can be a Subscriber and Publisher at once.
In our framework topics are defined by the messages types.

In our framework there are two different types of messages:

1. **Events**:

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

- Each Event expects a result of type Future<T>. This result should be resolved once the Subscriber which handles the Event completes processing it. For example: the AgentsAvailableEvent should return an object that states whether or not a certain 00 agent (or a couple of them) is available to execute a mission (e.g. both *007 James Bond* and *006 Alec Trevelyan* took part in the execution of the mission *Thunderball*). Some more information might be specified on the resolved Future<T> object. The result type is represented in the template Future object (T defines the result type).

- While a Subscriber processes an event, it might create new events and publish them to the MessageBroker. The MessageBroker will then assign the new events to the queue of one of the appropriate Subscribers. For example: while processing an MissionReceivedEvent, the Subscriber processing the event may need to check if a specific gadget is available in the inventory (e.g. in *Thunderball*, *007* used a *Sky Hook*). Therefore, it must create an event for checking the availability of this gadget in the inventory. The new event will be processed by another Subscriber which has access to the inventory. In case the Publisher which generated the new event is interested in receiving the result of the new event in order to proceed, it should wait until the new event is resolved (the computation completed) and retrieve its result from the Future object.

2. **Broadcast**:

- Broadcast messages represents a global announcement in the system. Each Subscriber can register to the type of broadcast messages it is interested to receive. The

MessageBroker sends the broadcast messages that are passed to it to all the Subscribers who are registered to the topic (this is in contrast to events - those are sent to only one of the relevant Subscribers).

In a round-robin manner we insert the event to one of the Subscribers registered to it in turns. For example, if two Subscribers - S1, S2 - are registered to handle events of type MissionReceivedEvent, and there are four events of this type – MissionReceivedEvent1, MissionReceivedEvent2, MissionReceivedEvent3, MissionReceivedEvent4, then MissionReceivedEvent1 will be handled by S1, MissionReceivedEvent~~1~~ - by S2, and after that we get back to MS1, so MissionReceivedEvent3 will be handled by S1, and so on.

4.2 EXAMPLE

In order to specifically explain how this framework operates, we'll (partly) describe the flow of our system. There are four types of Subscribers:

- Intelligence: creates a MissionReceivedEvent, which holds a mission name (e.g. *Thunderball*), an agent number (e.g. *007*.), and a gadget (e.g. *Sky Hook*).
- Q: this guy has an access to the inventory, where he keeps the gadgets he invented for the 00 agents.
- M: she's the boss. She handles a MissionReceivedEvent (note that in the assignment there might be several instances of M, in contrast to the traditional series, in which M is a singleton).
- Moneypenny - she checks the availability of 00 agents (as above, in the assignment there might be several instances of Moneypenny).

In addition, there are three types of messages:

- MissionReceivedEvent
- AgentsAvailableEvent
- GadgetAvailableEvent
- TickBroadcast

4.2.1 INITIALIZING A SUBSCRIBER

- Each Subscriber should register itself to a topic in the MessageBroker and initialize itself. During initialization, the Subscriber subscribes to the messages that it is interested to receive. Since M handles MissionReceivedEvent, she should subscribe to receive messages of that type. Similarly, as Q is interested in GadgetAvailableEvent, he should subscribe to receive those messages.

- When a Subscriber subscribes for a certain message, it supplies a callback which defines how the Subscriber should proceed upon receiving this specific type of message. This callback will be called by the Subscriber to process the message.

Figure 1 describes the initialization phase:

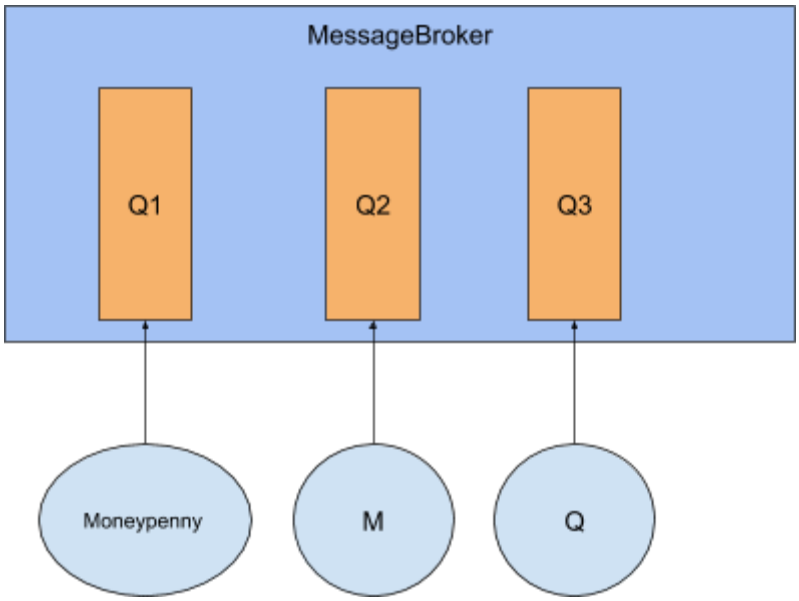
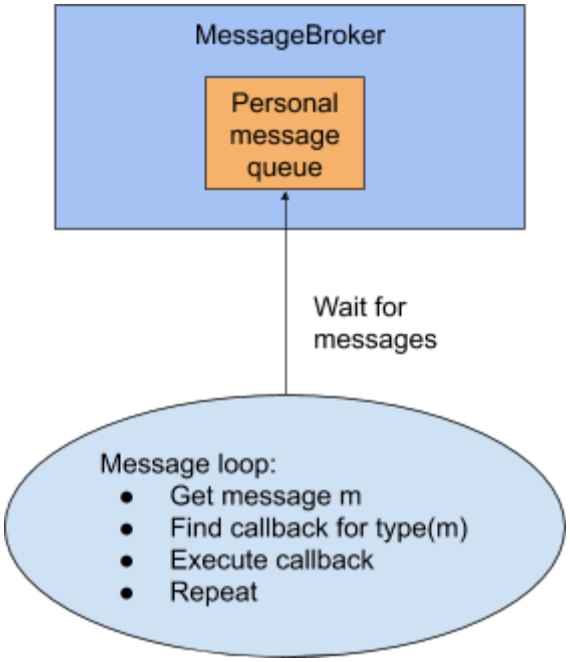


Figure 1

4.2.2 MESSAGE LOOP PATTERN

In this part you will implement the Message Loop design pattern. In such pattern, each Subscriber is a thread which runs a loop. In each iteration of the loop, the thread tries to fetch a message from its queue and process it.
An important point in such a design pattern is not to block the thread for a long time, unless there is no messages for it. Figure 2 describes the Message Loop in our system.

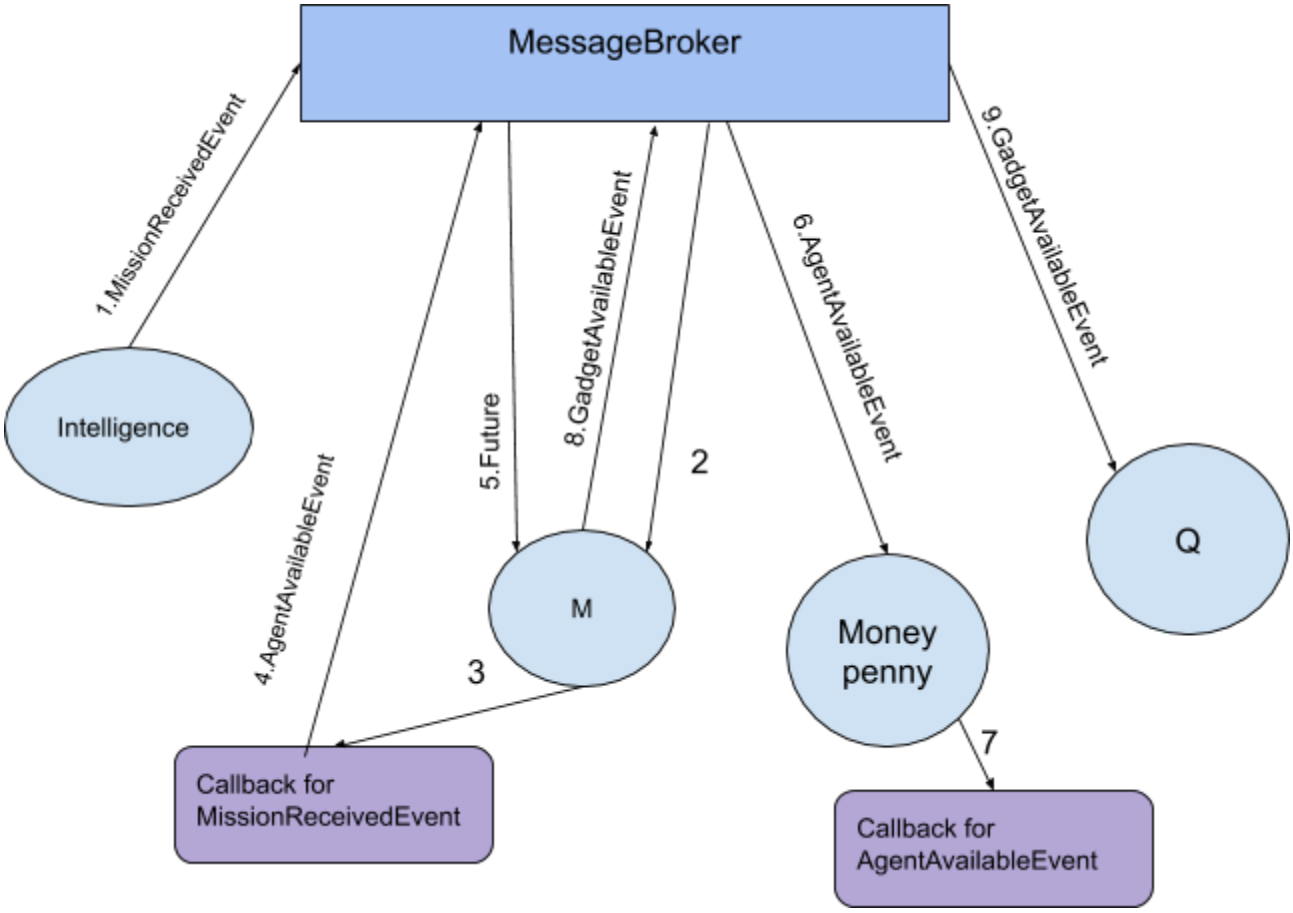


4.2.3 PROCESSING EVENTS

Figure 3 describes the interaction between the Subscribers and the MessageBroker:

1. Intelligence sends a MissionReceivedEvent to the MessageBroker.
2. The MessageBroker inserts this event to one of the M-Subscribers.
3. M fetches the event (in the message loop) and the corresponding callback is called to handle MissionReceivedEvent.
4. In the callback, there is a need to check the availability of certain agents. Since M has no access to the squad (where a collection of 00 agents are stored), she sends an AgentsAvailableEvent to the MessageBroker.
5. The MessageBroker receives the AgentsAvailableEvent and returns a Future object to M, who waits until this Future object is resolved.
6. The MessageBroker delivers the AgentsAvailableEvent to Moneypenny (who has access to the squad).
7. Moneypenny accepts the AgentsAvailableEvent and the corresponding callback processes it and resolves the Future object accordingly.
8. Now that the Future object is resolved, M looks for a gadget and sends a GadgetAvailableEvent.
9. The MessageBroker insert the GadgetAvailableEvent to the queue of Q, who will process the event and return an answer.

IMPORTANT - this is only a partial flow. The complete mechanism is reacher in details and will be fully described later on.



4.3 IMPLEMENTATION INSTRUCTIONS

To this end, each Subscriber\Publisher in our framework will run on its own thread. The different Subscribers\Publishers will be able to communicate with each other using only a shared object - the MessageBroker. The MessageBroker supports sending and receiving of two types of events: Broadcast messages (which upon being sent are delivered to every subscriber of the specific message type), and Event messages (which upon being sent are delivered to only one of its subscribers in a round-robin manner). The different Subscribers will be able to subscribe for message types they would like to receive using the MessageBroker. The different Subscribers\Publishers do not know of each other's existence. All they know of are the messages that are received in their message-queue which is located in the MessageBroker.

When building a framework, one should change the way they think. Instead of thinking like a programmer who writes software for end users, they 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. For this part of the assignment you will build a framework (write

code for other programmers), and the programmer which will use your code in order to develop its application will be the future you while 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 going 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 MessageBroker, Subscriber and the Publisher at this part. The following is a summary and additional clarifications about the implementation of different parts of the framework.

- Message: a data-object which is passed between Subscribers\Publishers 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 MessageBroker, it will be received by all the subscribers of this Broadcast-message type.
- Event: a marker interface extending Message. A Publisher that sends an event message may expect to be notified when the Subscriber that processes the event has completed processing it. The event has a generic type variable T, which indicates its expected result type (should be passed back to the Publisher). The Subscriber that has received the event must call the method 'Complete' of the MessageBroker once it has completed treating the event, in order to resolve the result of the event.
- SimplePublisher: a simple class that publishes messages (Events and Broadcasts) to the MessageBroker, any class that has SimplePublisher as an instance becomes a Publisher, we'll allow only the abstract classes Subscriber and Publisher to hold an instance of a SimplePublisher.
- Publisher: an abstract class that each Publisher in the system must extend. A publisher that extend this class should supply the initialize method and the run method (the Publisher is a Runnable).
- Subscriber: an abstract class that each Subscriber in the system must extend. The abstract Subscriber class is responsible to get and manipulate the singleton MessageBroker instance. Derived classes of Subscriber should never directly deal with the MessageBroker. Instead, they have a set of internal protected wrapping methods they can use. When subscribing to message types, the derived class also supplies a callback function. The Subscriber stores this callback function together with the type of the message it is related to. The Subscriber is a Runnable (i.e. suitable to be executed in a thread) - its run method implements a message loop. When a new message is taken from the queue, the Subscriber invokes the appropriate callback function.
  When the Subscriber starts executing the run method, it registers itself with the MessageBroker, and then calls the abstract initialize method. The initialize method allows derived classes to perform any required initialization code (e.g. subscribe to messages). Once the initialization code is completed, the actual message-loop should start. The Subscriber should fetch messages from its message queue using the MessageBroker's 'awaitMessage' method. For each message it should execute the corresponding callback. The Subscriber class also contains a terminate method that

should signal the message-loop that it should end. In our system every Subscriber is **also** a publisher since it has an access to SimplePublisher .

IMPORTANT:

- All the callbacks that belong to the Subscriber must be executed inside its own message-loop.
- Registration, initialization, and unregistration of the Subscriber must be executed inside its run method.

- MessageBroker: A shared object used for communication between Publishers and Subscribers. It should be implemented as a thread-safe singleton, as it is shared between all the Publishers and Subscribers in the system. The implementation of the MessageBroker interface should be inside the class MessageBrokerImpl (provided to you). There are several ways in which you can implement the MessageBroker methods; be creative and find a good, correct and efficient solution. **Notice, fully synchronizing this class will affect all the Subscribers\Publishers in the system (and your grade) - try to find good ways to avoid blocking threads as much as possible.**
  The MessageBroker manages the queues of the Subscribers. It creates a queue for each Subscriber using the 'register' method. When the Subscriber calls the 'unregister' method of the MessageBroker, the MessageBroker should remove its queue and clean all references related to that Subscriber. Once the queue is created, a Subscriber can take the next message in the queue using the 'awaitMessage' method. The 'awaitMessage' method is blocking, that is, if there are no messages available in the Subscriber queue, it should wait until a message is available.

  MessageBroker methods explained:
  - register: a Subscriber calls this method in order to register itself. This method should create a queue for the Subscriber in the MessageBroker.
  - subscribeEvent: A Subscriber calls this method in order to subscribe itself for some type of event (the specific class type of the event is passed as a parameter).
  - subscribeBroadcast: A Subscriber calls this method in order to subscribe itself for some type of broadcast message (The specific class type of the broadcast is passed as a parameter).
  - sendBroadcast: A Publisher calls this method in order to add a broadcast message to the queues of all Subscribers which subscribed to receive this specific message type.
  - Future sendEvent(Event e): A Publisher calls this method in order to add the event e to the message queue of one of the Subscribers which subscribed to receive events of type e.getClass(). The messages are added in a round-robin manner. This method returns a Future object - from this object the sending Publisher can retrieve the result of processing the event once it is completed. If there is no suitable Subscriber, the method should return null.

- void complete(Event e, T result): A Subscriber calls this method in order to notify the MessageBroker that the event was handled, and providing 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 Subscriber calls this method in order to unregister itself. Should remove the message queue allocated to the Subscriber and clean all the references related to this MessageBroker.
- awaitMessage(Subscriber s): A Subscriber calls this method in order to take a message from its allocated queue. This method is blocking (waits until there is an available message and returns it).

5. PART 2 - THE FULL SYSTEM

5.1 OVERVIEW

In this part you will build a system *In Her Majesty's Secret Service*. In order to build this system, you will use the Pub-Sub framework from part 1.
NOTE: The system described in this part is reacher in details from the one that was given above as an example. Please pay attention.
The end-to-end description of our system is simple. Different intelligence sources will deliver info and ask for a 00 agent (or a couple of them) and a gadget that are needed for some mission. One of the M-subscribers will process this event, it must know whether the 00 agent is available in the squad (M cannot perform that check on its own. She will send an event that will be handled by one of the Moneypenny-subscribers), and whether a fit gadget in his Inventory (as before, M cannot perform that check on its own. She will send an event that will be handled be Q). After those checks are made, M fills a report to her diary in case all conditions are met. She also increments a counter of total received missions (this counter is incremented whether the mission was executed or not). This should happen while communication between the Subscribers\Publishers is allowed only via the MessageBroker.

5.2 PASSIVE OBJECTS

This section contains the data classes (a.k.a. non-runnable classes) which you need to implement. The name of the class appears first, then a short description if needed, the mandatory fields, and the mandatory methods (getters and setters are also needed, but will not be specified):

- Agent - simply, a 00 agent.
  - String serialNumber - eg. 006, 007, 008, 0012 (James Bond 007 is not the only 00, though he's the famous of all). serialNumber is represented by a String (not an int), to allow the 00 prefix… this is super important.
  - String name - e.g. James Bond, Bill Fairbanks, Sam Johnston (007, 002, 0012 respectively. FYI).
  - Boolean available - initialized with 'true'. May be changed to 'false' by a Moneypenny-instance, then changed to 'true' again when a mission is completed/aborted.
  - void acquire() - changes 'available' flag from 'true' to 'false'.
  - void release() - changes 'available' flag from 'false' to 'true'.

- Squad (singleton) - a collection of 00 agents.
  - Map<String, Agent> agents - where a serialNumber is the key, and an agent - the value.
  - Boolean getAgents(List<String> serials) - return 'false' if one of the agents of serialNumber 'serial' is missing, and 'true' otherwise (should acquire() the agents before, in addition). If an agent is in the Squad, but is already acquired for some other mission, the function will wait until the agent becomes available.
  - void releaseAgents(List<String> serials) - releases the agents with the specified serials (in case mission is aborted)
  - List<String> getAgentsNames(List<String> serials) - returns a list of the names of the agents with the specified serials.
  - void sendAgents(List<String> serial, int timeTick) - simulates executing a mission by calling sleep() (the required sleep-time - in time-ticks! - is given as parameter), and then calls the release() method of the agents with the specified serial numbers.
  - void load (Agent[] agents) - initialization method for the Squad.

- Gadget - simply a String (e.g. *Sky Hook*, *Explosive Pen*).

- Inventory (singleton) - that's where Q holds his gadget.
  - List<String> gadgets.
  - Boolean getItem(String gadget) - return 'false' if the gadget is missing, and 'true' otherwise. The method should **permanently** remove() the gadget from the list too

(as traditionally, a 00 agent never brings back a gadget in one piece… For simplicity, the gadget will be removed even in a case were the mission will be eventually aborted).
    - printToFile(String filename) - prints all of the reports to a json file.
    - void load (String[] gadgets) - loads the gadgets to the inventory in the initialization phase.

- MissionInfo - information about a mission.
    - String missionName.
    - List<String> serialAgentsNumbers.
    - String gadget.
    - int timeIssued - in this time-tick, the mission will be sent by the Intelligence subscriber.
    - int timeExpired - if this time-tick is reached before execution of a mission, it won't be executed at all.
    - int duration - duration of the mission in time-ticks. this parameter will be passed to the send(int time) function of the required agent.

- Report - in which mission details are specified. The following details are required:
    - String mission name.
    - int M - remember that in our assignment there are several M-instances. They differ by serial number. The M instance who handled the mission will assigns its serial number to this parameter.
    - int Moneypenny - same as above. The Moneypenny-instance who asks for agents for this mission will have its serial number here.
    - List<String> agentsSerialNumbers
    - List<String> agentsNames
    - String gadgetName
    - int timeIssued - the time-tick when the mission was sent by an Intelligence Publisher.
    - int QTime - the time-tick in which Q Received the GadgetAvailableEvent for that mission..
    - int timeCreated - the time-tick when the report has been created.

- Diary (singleton) - where all reports are stored.
    - List<Reports> reports - only executed missions will be reported here.
    - int total - total number of received missions (executed / aborted) be all the M-instances.
    - printToFile(String filename) - prints all of the reports and the number of received missions (int total) to a json file.
    - void addReport(Report reportToAdd) - adds a report to the diary.

5.3 MESSAGES

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

- **MissionReceivedEvent**
- **AgentsAvailableEvent**
- **GadgetAvailableEvent**
- **TickBroadcast -** A broadcast messages that is sent by the TimeService at every passed time-tick. It will be described below.

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

5.4 ACTIVE OBJECTS (Subscribers/Publishers)

Remember: Subscribers\Publishers MUST NOT know of each other. A Subscriber\Publisher must not hold a reference to other Subscribers\Publishers, neither get a reference to another Subscriber\Publisher.

- **TimeService** (There is only one instance of this Publisher).

This Publisher is our global system timer (handles the clock ticks in the system). It is responsible for counting how much clock ticks passed since its initial execution and notify every other Subscriber (that is interested) about it using the TickBroadcast.
The TimeService sends a time-tick every 100 milliseconds, and takes as a constructor argument (int duration) the number of **ticks** before termination.
The TimeService stops sending TickBroadcast messages after the duration ends.
(See the Java Timer class. It's helpful)

- **Intelligence**

Holds a list of Info objects (described above). Each Info has a timeIssued member. When this time-tick is reached, Intelligence Publisher Sends a MissionReceivedEvent with the information that is needed to accomplish the mission.

- **Moneypenny**

Only this type of Subscriber can access the squad. Three are several Moneypenny-instances - each of them holds a unique serial number that will later be printed on the report.

- **Q** (There is only one instance of this Subscriber).

Q is the only Subscriber that has access to the inventory. The time-tick in which Q receives a GadgetAvailableEvent will be printed in the report (if the mission was eventually executed).

- **M**

M handles MissionReceivedEvent.
For simplicity, when M-instance receives such an event, it should do the tasks in the following order: first it tries to acquire the necessary agents, then it checks the availability of the required gadget, and only then it checks if the mission's expiry time had reached. If not - then M acknowledges Moneypenny to Agent.send() the required agents, and a report will be added to the diary (if expiry time passed, M orders Moneypenny to release() ll agents).
In any case (mission either executed or aborted), Diary.total will be incremented.
Each of the M instances holds a unique serial number that will later be printed on the report.
Please follow the order specified above. Otherwise, your output may be inconsistent with the required output.

Very Important:
Only Q knows about the inventory, only Moneypenny-instances know about the squad, only M-instances know about the diary. All communication between the Publishers/Subscribers is done via messages and Future objects that are sent/received to/from the MessageBroker.

Tips for implementation:

1. Make sure you understand the flow. You can draw a diagram that describes the events exchange between the Subscribers\Publishers since a mission is received, until it is completed.
2. Use blocking method carefully.
3. When several 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?

5.5 INPUT FILES AND PARSING

5.4.1 THE JSON FORMAT

All your input files for this assignment will be given as JSON files. You can read about JSONs syntax here.
In Java, there are a number of different options for parsing JSON files. Our recommendation is to use the 'Gson' library. See the Gson User Guide and APIs, there are a lot of informative examples there.

5.4.2 EXECUTION FILE

Attached to this assignment is an execution file example. The file holds a json object which contains the following fields:

For passive objects:

- Inventory: An array of the gadgets in the inventory.
- Squad: An array of the 00 agents.

For Subscribers\Publishers:

- Intelligence: An array(1) of arrays(2). Array(1) represent an intelligence source, while in array(2) you'll find the missions it will send (as events) to the MessageBroker. For every mission, there will be a time parameter - that one determines the time-tick in which the Publisher is supposed to send a corresponding event.
- Moneypenny: the number of Moneypenny instances will be given as a parameter. Remember - these instances differ by serial number that will be printed to the reports.
- M: same as above.
- Time: The termination tick will be given as a parameter.

5.6 OUTPUT FORMAT

Your program should output the following two json objects, each one in a different file. The names of the output files are given as command line arguments (see next section). The json files you will generate represent the two objects:

- List<String> the gadgets that are left in the inventory when the program terminates.
- Diary.total followed by List<Report> from the diary of M.

5.7 PROGRAM EXECUTION

The SystemRunner class will run the simulation. When started, it should accept as command line argument the name of the json input file and the names of the two output files - the first file is the output file for the gadgets, and the second is for the List<Report> - the **exact** order of the parameters should be:

1. "inputFile.json"
2. "inventoryOutputFile.json"

3. “diaryOutputFile.json”

The SystemRunner should read the input file, Next it should create the passive objects (Inventory, Squad of Agents), and finally - create and initialize the Subscribers and Publishers. At a certain time-tick, the program should terminate - all Subscribers should unregister themselves and terminate gracefully (that’s important!). After that happens, the SystemRunner should generate the output files and exit.

5.8 JUnit Tests

Testing is an important part of developing. It helps us make sure our project behaves as we expect it to. This is why in this assignment, you will use Junit for testing your project. You are required to write unit tests for the classes in your project. This must be done for (at least) the following classes:

- MessageBroker
- Future
- Inventory
- Squad

You need to submit the unit tests by 12.12.2019 (before the deadline of the assignment).

5.9 TESTING YOUR APPLICATION

- You should write input to test your application. We supplied you with one input file which you can use as an example - but please test your application with input files of your own, and check different scenarios. We will run your program on several input files.
- The output of your program is several serialized objects, that is, the output files you generate includes binary data. You have to de-serialize these files in order to get the actual objects. After you get the actual objects, you can run the methods of the class on them (we need the getters methods).
- What to check:
    - Correctness - if all of the required items for a mission (agents, gadgets) are available and the current time is legal the mission should be executed.
    - Consistency: few examples.
      number of sent mission >= Diary.total >= Diary.reports.size() >= 0.
      Squad.getAgent() returns true only if the agent is available. In addition, after this function returns ‘true’, Agent.available = false.

Inventory.getItem() returns true only if the item exists. In addition, after this function returns ‘true’, Gadget.available = false.
For any given executed mission, Report.time <= Mission.timeExpired.

- Round-Robin scheduling - the events are processed by the Subscribers in round-robin, e.g. if there are 5 events of MissionReceivedEvent and 5 M-instances, we expect each M to process one event.

5.10 BUILDING THE APPLICATION: MAVEN

In this assignment you are going to be using maven as your build tool. Maven is the de-facto java build tool. In fact, maven is much more than a simple build tool, it is described by its authors as a software project management and comprehension tool. You should read a little about how to use it properly. IDEs such as Eclipse, Netbeans and Intellij all have native support for maven and may simplify interaction with it - but you should still learn yourself how to use it. Maven is a large tool. In order to make your life easier, you have (in the code attached to this assignment) a maven pom.xml file that you can use to get started - you should learn what this file is and how you can add dependencies to it (and what are dependencies).

5.11 SUBMISSION INSTRUCTIONS

5.10.1 FILE STRUCTURE

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

5.10.2 SUBMISSION

- Submit all the package (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.
- **You must submit one ‘.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 majeek at cs. Your request email must include the following information:
    - Your name and your partner's name.
    - Your id and your partners id.
    - Explanation regarding the reason for the extension request.
    - Official certification.
- **Request without a compelling reason will not be accepted.**

5.12 GRADING

Although you are free to work wherever you please, 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. "But it worked fine on my windows-based home computer" will not earn you any mercy points from the grading staff if your code fails to execute at the lab. Grading will tackle the following points:

- Your application design and implementation.
- Automatic 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 of synchronization and unnecessary interference with the liveness of the program.
- Checking if your implementation follows the guidelines detailed in “Documentation and Coding Style”.