SPL191- ASSIGNMENT 2

Java Generics, Concurrency, and Synchronization

TAs in charge:

Hussien Othman

Yair Ashlagi

Publication date: **21.11.2018**

Deadline: **11.12.2018 23:59**

Unit tests submission deadline: **28.11.2018 23:59 (see section 5.7).**

0. 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 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 the 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 Micro-Service framework, which you will then use to implement an online book store with a delivery option. The Micro-Services architecture has become quite popular in recent years. In the Micro-Services architecture, complex applications are composed of small and independent services which are able to communicate with each other using messages. The Micro-Service 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 Micro-Service framework.
2. Implementing an online books store application on top of this framework.

It is very important to read the entire work before starting. Do not be lazy here, the work will be much easier if you read and understand the entire work in advance.

3 PRELMINARY

In this section you will implement a basic Future class which we will use during the rest of the assignment. A Future object represents a promised result - an object that will eventually be resolved to hold a result of some operation. 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 available.

4 PART 1: SYNCHRONUS MICROSERVICES FRAMEWORK

4.1 DESCRIPTION

In this section we will build a simple Micro-Service framework. A Micro-Service framework consists of two main parts: A Message-Bus and Micro-Services. Each Micro-Service is a thread that can exchange messages with other Micro-Services using a shared object referred to as the Message-Bus. There are two different types of messages:

Events:

- An Event defines an action that needs to be processed, e.g., ordering a book from a store. Each Micro-Service specializes in processing one or more types of events. Upon receiving an event, the Message-Bus assigns it to the messages queue of an appropriate Micro-Service which specializes in handling this type of event. It is possible that there are several Micro-Services which specialize in the same events, (e.g., two or more Micro-Services which can handle an event of ordering a book from a books store). In this case, the Message-Bus assigns the event to a single Micro-Service of those which specialize in this event in a round-robin manner (described below).

- Each event holds a result. This result should be resolved once the Micro-Service which the event was assigned to completes processing it. For example: the event of 'ordering a book' from the store should return a receipt in the case that the order was successfully completed. The result is represented in the template Future<T> object (T defines the type of the results).

- While a Micro-Service processes an event, it might create new events and send them to the Message-Bus. The Message-Bus will then assign the new events to the queue of one of the appropriate Micro-Services. For example: while processing an 'ordering a book' event, the Micro-Service processing the event may need to check if the specific book is available in the inventory. Therefore, it must create an event for checking the availability in the inventory. The new event will be processed by another Micro-Service which has access to the inventory (such as the Manager for instance). In case the Micro-Service which generated the new event is interested in receiving the result of the new event in order to proceed, therefore it should wait until the new event is resolved (the computation completed) and retrieve its result from the Future object.

Broadcast: Broadcast messages represents a global announcement in the system. Each Micro-Service can subscribe to the type of broadcast messages it is interested to receive. The Message-Bus sends the broadcast messages that are passed to it to all the Micro-Services which are interested in receiving them (this is in contrast to events which are sent to only one of the Micro-Services which are interested in them).

4.1.1 ROUND ROBIN PATTERN

In a round robin manner we insert the event to one of the micro-services subscribed to receive it in round order. For example: if for event of type OrderBook subscribe 4 services in order to receive it: ServiceA, ServiceB, ServiceC, ServiceD, and there are 6 orders – OrderBook1, OrderBook2, OrderBook3, OrderBook4, OrderBook5, OrderBook6. Then: OrderBook1 will be inserted to ServiceA, OrderBook2 to ServiceB, OrderBook3 to ServiceC and OrderBook4 to ServiceD. After that we get back to ServiceA, so OrderBook5 will be inserted to ServiceA and OrderBook6 to ServiceB, and so on.

4.2 EXAMPLE

In order to specifically explain how this framework operates, assume our goal is to write a small application for an online book store. The book store in our example consists of 3 types of services:

- WebAPI
- SellingService
- StoreManager

In addition, there are three types of messages:

- OrderBookEvent: (Event) represents a book order that should be processed.
- CheckAvailability: (Event) represents an event for checking if a book exists in the inventory.
- FiftyPercentDiscount: (Broadcast) represents a announcement that there is a discount on some specific books.

4.2.1 FUNCTIONALITY OF EACH MICRO-SERVICE

StoreManager: Decides on advertising discounts in order to promote sales. The discount messages are sent to all the Micro-Services of types WebAPI and SellingService which registered to these discount broadcasts and are interested in receiving them. In addition, it specializes in retreiving books from the inventory meaning that it processes events of type 'CheckAvilability'.

WebAPI: The WebAPI (representing a connection with a store customer) may send an OrderBookEvent on behalf of a real human customer that is logged in to the store website. Also, it is interested in discount messages in order to inform the customer about new discounts. The store creates a single WebAPI service per logged in customer in order to support a large number of clients.

SellingService: Handles the OrderBookEvent. While processing an OrderBookEvent, it creates a CheckAvilability event. If the book is in the inventory, it confirms the order and returns receipts. Also, it is interested in receiving discount messages.

4.2.2 INITIALIZE MICRO-SERVICE

- Each Micro-Service should register itself to the Message-Bus and initialize itself. During initialization, the Micro-Service subscribes to the messages that it is interested to receive. Since the SellingService is handling OrderBookEvent, it should subscribe to receive OrderBookEvent messages. Similarly, since the discounts which the manager decides on matter to the SellingService and the WebAPI services, both should subscribe to receive FiftyPercentDiscount messages.
- When A Micro-Service subscribes for a certain message, it supplies a callback which defines how the Micro-Service should proceed upon receiving this specific type of message. This callback will be called by the Micro-Service to process the message.

Figure 1 describes the initialization phase.

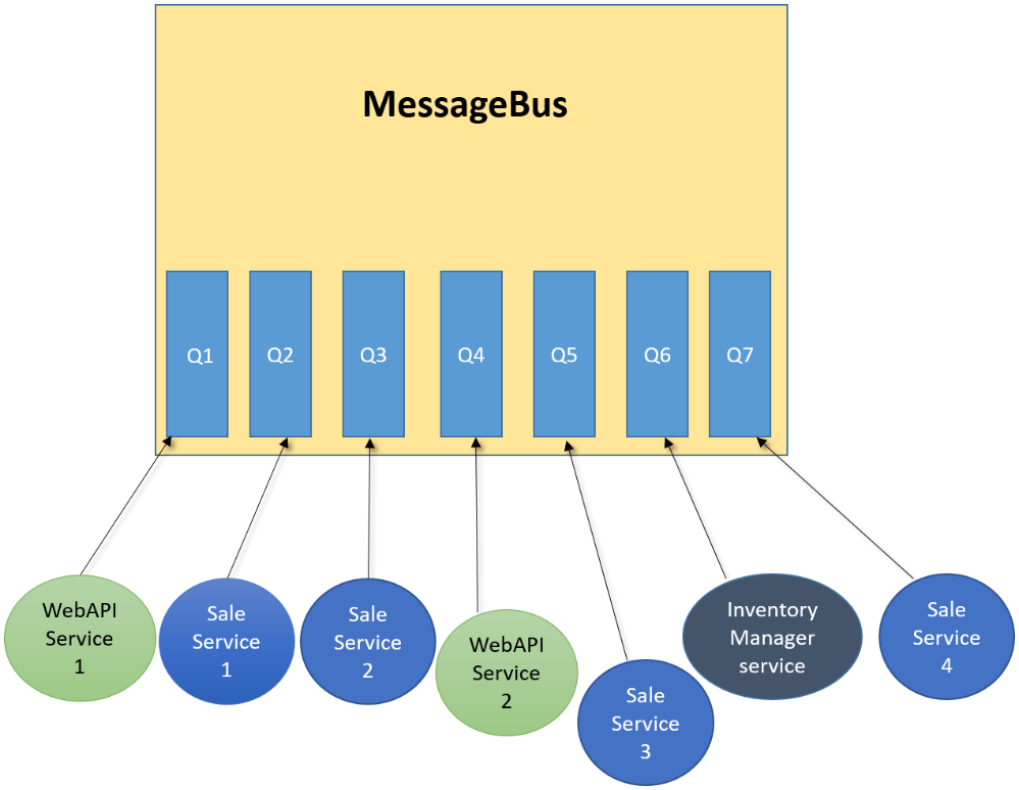

Figure 1

4.3 MESSAGE LOOP PATTERN

In this part you have to implement the Message Loop design pattern. In such pattern, each micro-service 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 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.

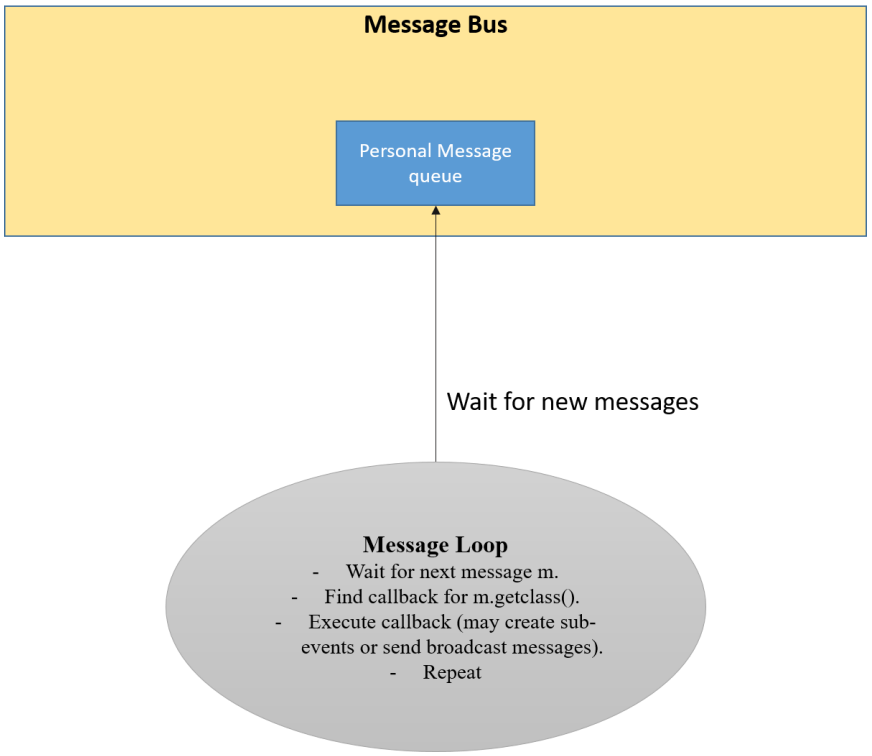

Figure 2

4.4 PROCESSING EVENTS

Figure 3 describes the interaction between the micro-services and the message-bus and between themselves.

1. The WebAPIService 1 sends an OrderBookEvent to the message-bus.
2. The message-bus inserts this event to one of the Selling Services, which is SellingService1.
3. SellingService 1 fetches the event (in its Message Loop) and calls the corresponding callback which handles OrderEvent.
4. In the callback, there is a need to check if the book is on inventory. Since SellingService 1 has no access to the Inventory, it sends an CheckAvailabilityEvent to the message-bus.
5. The message-bus receives the CheckAvailabiltyEvent and returns Future object to SellingService 1 which waits until this Future object becomes resolved.
6. The message-bus inserts the CheckAvailabiltyEvent the InventoryManagerService (which has an access to the inventory).

7. InventoryManagerService calls the corresponding callback to process CheckAvailabiltyEvent.
8. The InventoryManagerService checks if the book is available and calls complete to resolve the result. The corresponding Future object is resolved, and now holds the actual result.

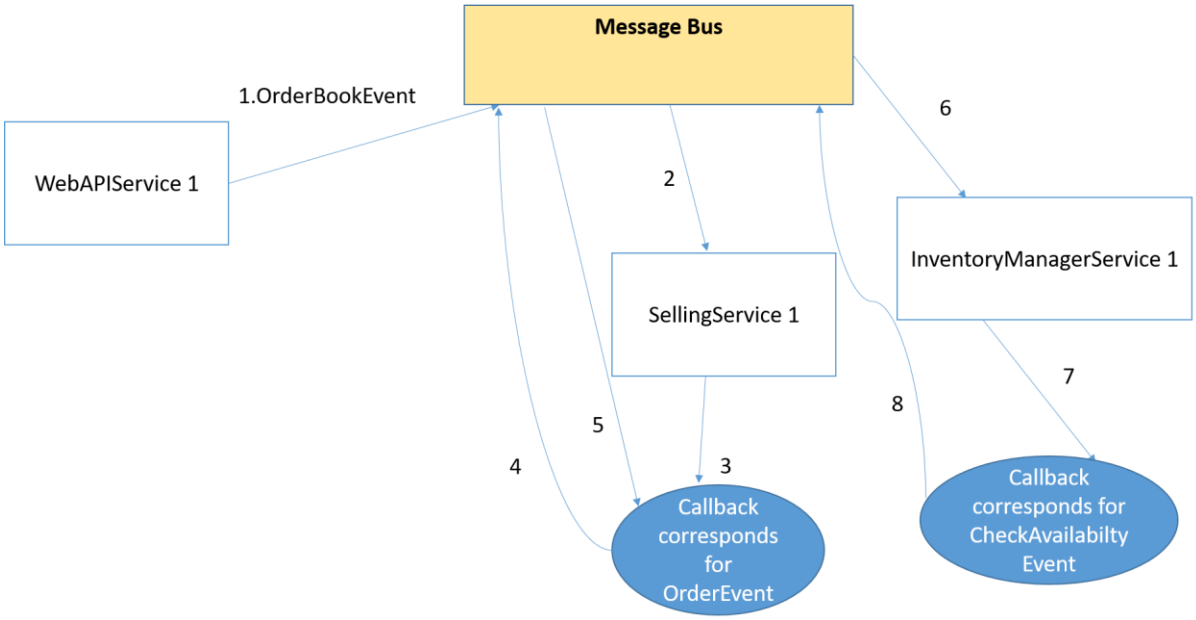

Figure 3

4.5 IMPLEMENTATION INSTRUCTIONS

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

When building a framework, one should change the way they think. Instead of thinking like programmer which writes software for end users, they should now think like a programmer writing a 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), the programmer which will use your code in order to develop its application will be the future you while they work on the second part of this 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 MessageBus and the MicroService for 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 Micro-Services as 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 received by all the subscribers of this Broadcast-message type (the message Class)
- **Event<T>:** A marker interface extending Message. A Micro-Service that sends an Event message expects to be notified when the Micro-Service 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 sending Micro-Service). The Micro-Service that has received the event must call the method 'Complete' of the Message-Bus once it has completed treating the event, in order to resolve the result of the event.
- **MessageBus:** The Message-Bus is a shared object used for communication between Micro-Services. It should be implemented as a thread-safe singleton, as it is shared between all the Micro-Services in the system. The implementation of the MessageBus interface should be inside the class MessageBusImpl (provided to you). There are several ways in which you can implement the message-bus methods; be creative and find a good, correct and efficient solution. Notice, **fully synchronizing this class will affect all the micro-services in the system (and your grade!) - try to find good ways to avoid blocking threads as much as possible.**

  The Message-Bus manages the queues of the Micro-Services. It creates a queue for each Micro-Service using the 'register' method. When the Micro-Service calls the 'unregister' method of the Message-Bus, the Message-Bus should remove its queue and clean all references related to that Micro-Service. Once the queue is created, a Micro-Service 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 Micro-Service queue, it should wait until a message becomes available.
  - register: a Micro-Service calls this method in order to register itself. This method should create a queue for the Micro-Service in the Message-Bus.
  - subscribeEvent: A Micro-Service 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 Micro-Service calls this method in order to subscribe itself for some type of broadcast message (The specific class type of the event is passed as a parameter).
  - sendBroadcast: A Micro-Service calls this method in order to add a broadcast message to the queues of all Micro-Services which subscribed to receive this specific message type.
  - Future<T> sendEvent(Event<T> e): A Micro-Service calls this method in order to add the event *e* to the message queue of one of the Micro-Services which have subscribed to receive events of type e.getClass(). The messages are added in a round-robin fashion. This method returns a Future object - from this object the sending Micro-Service can retrieves the result of processing the event once it is completed. If there is no suitable Micro-Service, should return null.
  - void complete(Event<T> e, T result): A Micro-Service calls this method in order to notify the Message-Bus 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 Micro-Service calls this method in order to unregister itself. Should remove the message queue allocated to the Micro-Service and clean all the references related to this Message-Bus.
  - awaitMessage(Microservice m): A Micro-Service 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).
- **MicroService:** The MicroService is an abstract class that any Micro-Service in the system must extend. The abstract MicroService class is responsible to get and manipulate the singleton MessageBus instance. Derived classes of MicroService should never directly touch the Message-

Bus. 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 MicroService stores this callback function together with the type of the message it is related to. The Micro-Service is a Runnable (i.e., suitable to be executed in a thread). The run method implements a message loop. When a new message is taken from the queue, the Micro-Service will invoke the appropriate callback function.

When the Micro-Service starts executing the run method, it registers itself with the Message-Bus, 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 completes, the actual message-loop should start. The Micro-Service should fetch messages from its message 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. Each Micro-Service contains a name given to it in construction time (the name is not guaranteed to be unique).

IMPORTANT:

- **All the callbacks that belong to the micro-service must be executed inside its own message-loop.**
- **Registration, Initialization, and Unregistration of the Micro-Service must be executed inside its run method.**

4.5.1 CODE EXAMPLE

You can find in the package *bgu.spl.mics.example* a usage example of the framework. This example is simple, it creates simple services: ExampleBroadcastListenerService, ExampleEventHandlerService, and ExampleMessageSenderService, and two messages: ExampleBroadcast and ExampleBroadcast. Attached to this document an example of testing your part 1 code.

5 PART 2: BUILDING AN ONLINE BOOK STORE

Important: see section 4.8 before you start implementing this part.

5.1 OVERVIEW OF OUR SYSTEM

In this part you will build an online book store management system. In order to build this system, you will use the Micro-Service framework from part 1. The end-to-end description of our system is simple. A customer connects to the store website and orders a book. If the book is available and the customer has enough credit, the order is confirmed - the customer pays for the book, and then the book should be delivered to his address as soon as possible.

NOTE: The book store in this part is different from the one described in the introduction of the assignment.

5.2 PASSIVE OBJECTS

This section contains a list of passive classes (a.k.a., non-runnable classes) which you need to implement:

- ***BookInventoryInfo:*** An object which represents information about a single book in the store. It contains the following fields:

- *bookTitle*: String – the title of the book.
- *amountInInventory*: int – the number books of *bookTitle* currently in the inventory.
- price: int – the price of the book.

- ***OrderReceipt:*** An object representing a receipt that should be sent to a customer after buying a book (when the customers OrderBookEvent has been completed). The receipt should contain the following fields:
  - orderId: int – the id of the order.
  - *seller*: string - the name of the service which handled the order.
  - *customer*: id - the id of the customer the receipt is issued to.
  - *bookTitle*: string – title of the book bought.
  - *price*: int – the price the customer paid for the book.
  - *issuedTick*: int - tick in which this receipt was issued (upon completing the corresponding event).
  - *orderTick*: int - tick in which the customer ordered the book.
  - proccessTick: int – tick in which the selling service started processing the order.
- ***Inventory:***
  This object must be implemented as a thread safe singleton. The Inventory object holds a collection of BookInventoryInfo: One for each book the store offers.
  Only the following methods should be publicly available from the store:
  - public void load (BookInventoryInfo[ ] inventory)
    This method should be called in order to initialize the store inventory before starting execution (by the BookStoreRunner class defined later). The method will add the items in the given array to the store.
  - public OrderResult take (String book)
    This method will attempt to take one book from the store. It receives the title of books to take. Its result is an enum which has the following value options:
    - NOT_IN_STOCK: which indicates that there were no books of this type in stock (the store inventory should not be changed in this case)
    - SUCCESSFULLY_TAKEN: which means that the item was successfully taken (the number books of this type should be reduced by one)
  - public int checkAvailabiltyAndGetPrice(String book): this method returns the price of the book if it is available, and -1 otherwise.
  - public void printInventoryToFile(String filename): prints to a file named *filename* a serialized HashMap<String,Integer> object, which is a map of all the books in the inventory. The key is the book's title and the value is the amount of this book in inventory. This method is called by the BookStoreRunner class in order to generate the output (described later).
- ***DeliveryVehicle:*** this object represents a delivery vehicle in the system. It consists of the following field:
  - license: int – the vehicle license number.
  - speed: int- number of milliseconds needed for 1KM.

  Contains the method *deliver* which gets as parameter the address of the customer and the distance from the store and simulates delivery by calling to *sleep* with the required number of milliseconds for delivery.

- ***MoneyRegister:***

This object holds a list of receipt issued by the store. This class should be implemented as a thread safe singleton. **Only** the following methods should be publicly available from the MoneyRegister:
    - public void file (OrderReceipt r): this method saves the given receipt in the MoneyRegister.
    - getTotalEarnings(): this method returns the total earnings of the store.
    - getOrderReceipts(): returns all the order receipts in the system.
    - printOrderReceipts(String filename): prints to a file named *filename* a serialized object List<OrderReceipt> which holds all the order receipts currently in the MoneyRegister. This method is called by the BookStoreRunner class in order to generate the output.

  Contains the method *chargeCreditCard* which gets as parameter the amount to charge from this customers credit card.
- ***ResourcesHolder:***
  Holds a collection of DeliveryVehicle. Only the following methods should be publicly available:
    - Future< DeliveryVehicle > acquireVehicle: tries to acquire a vehicle. Returns a Future object which will hold the acquired vehicle at some point.
    - void releaseVehicle: releases a vehicle, making it available to acquire again.
- ***Customer:***
    - id: int – The id number of the customer.
    - name: String – The name of the customer.
    - address: String – The address of the customer.
    - distance: int – the distance of the customer's address from the store.
    - Receipts: List – all the receipts issued to the customer.
    - creditCard: int – The number of the credit card of the customer.
    - availableAmountInCreditCard: The remaining available amount of money in the credit-card of the customer.

Messages:

The following message classes are **mandatory**, the fields that these messages hold are omitted; you need to think about them yourself:

Note: the micro-services are explained in the next section.

- BookOrderEvent:
  - An event that is sent when a client of the store wishes to buy a book. Its expected response type is an OrderReceipt. In the case that the order was not completed successfully, null should be returned as the event result.
  - Processing: if the book is available in the inventory then the book should be taken, and the credit card of the customer should be charged. If there is not enough money in the credit card, the order should be discarded, and the book should not be taken from the inventory.
  - Sent by the WebAPI service, the WebAPI waits for this event to complete to get the result. The event is sent to a SellingService to handle it.
    Note: there might be several orders of the same customer processed concurrently by different micro-services.
- TickBroadcast:
  A broadcast messages that is sent at every passed clock tick. This message must contain the current tick (int).
- DeliveryEvent

- An event that is sent when the BookOrderEvent is successfully completed and a delivery is required.
- Processing: should try to acquire a delivery vehicle. If the acquisition succeeds, then should call the method *deliver* of the acquired delivery vehicle and wait until it completes. Otherwise, should wait the vehicle becomes available.
- It is sent to the LogisticsService once the order is successfully completed. The sender does not need to wait on the event since it does not return a value.

Important note: You may create new types of messages as you see fit.

5.3 ACTIVE OBJECTS (MICROSERVICES)

Remember: Micro-services MUTS NOT know each other. A micro-service must not hold a reference to other micro-services, neither get a reference to another micro-service. If you do not fulfill this instruction, your grade will suffer from a significant reduction.

- **TimeService: (There is only one instance of this service).**
  This Micro-Service 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 Micro-Service (that is interested) about it using the TickBroadcast. The TimeService receives the number of milliseconds each clock tick takes (speed: int) together with the number of ticks before termination (duration: int) as constructor arguments. The TimeService stops sending TickBroadcast messages after the duration ends. Be careful that you are not blocking the event loop of the timer Micro-Service. You can use the Timer class in java to help you with that. The current time always start from 1.
- **APIService:**
  This Micro-Service describes one client connected to the application. The APIService expects to get the following arguments to its constructor:
  - orderSchedule: List – contains the orders that the client needs to make (every order has a corresponding time tick to send the OrderBookEvent). The list is not guaranteed to be sorted. The APIService will send the OrderBookEvent on the tick specified on the schedule (each order contains one book only, that is, orders on the same tick are supposed to be processed by different SellingService (in case there is more than one).
- **SellingService:**
  This Micro-Service handles OrderBookEvent. It holds a reference MoneyRegister object.
- **ResourceService:** this Micro-Service holds a reference to the *ResourcesHolder* instance.
- **InventoryService:** this Micro-Service holds a reference to the Inventory instance.
- **LogisticsService:** this Micro-Service handles the DeliveryEvent.

Very Important:

- Reference to Inventory object (and the books collection) should be held ONLY in InventoryService class only and they must NOT be returned to a caller.
- Reference to ResourcesHolder object must be held only in ResourceService class and it must NOT be returned to a caller.
- Reference to MoneyRegister should be held only in SellingService and it must NOT be returned to a caller.

- Note: You can create any new messages you need (Events and Broadcast). You can create any sub-events you need in order to implement the main events. You should decide which Micro-Service should handle the sub-events you create (Remember: you must avoid deadlocks!).
- **There might be any number of instances (>=1) of the different micro-service (expect the TimeService). That is, there might be one or more SellingService, ResourceService, InventoryService, LogisticsService, and APIService.**

Tips for implementation:

1. Make sure you understand all the events need to be processed.
2. Draw a diagram of the system (along with the events exchanged between the micro-services).
3. If service A waits for service B, and service B waits for service A – then a deadlock might occur. Identify all potential cases of deadlocks in the system and try to avoid them.
4. Use blocking method carefully.
5. 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 this? Where do not you need this?

5.4 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 to see how to work with Gson. There are a lot of informative examples.

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:

- initialInventory: An array containing books together with the following information for each book: amount that should be in the inventory when execution starts and their price.
- initialResources: An array containing the vehicles in the system. The information for a vehicles is its license number and speed.
- services: An object describing the Micro-Services that should run on the execution. It contains the following fields:
    - time: an object containing the arguments for the TimeService constructor. Its name is "time".
    - selling: a number representing the number of SellingServices to start with. Each SellingService started will carry the unique name 'selling i', where i starts from 1 to the total number of sellers (i.e., if sellers = 2, there will be 2 new SellingServices - one with the name "selling 1" and one with the name "selling 2").
    - inventoryService: a number representing the number of inventoryService to start with. The names like in SellingServices.
    - logistics: a number representing the number of LogisticsServices to start with. The names like in SellingServices.

- resourcesService: a number representing the number of ResourceServices to start with. The names like in SellingServices.
- customers: An array in which each item represents the constructor arguments for a WebAPI service that needs to be initialized.

5.5 OUTPUT FORMAT

Your program should output the following ***serialized*** objects, each one in a different file. The names of the output files are given as arguments in the command line (see next section): (you are allowed to add *implements serializable* to the classes).

- **HashMap<Integer,Customer>** which contains all the customers in the system. The key is the id of the customer and the value is the reference of the customer instance.
- **HashMap<String,Integer>** which contains all the books in the inventory and their remained amount. The key is the books title and the value is the reference of the customer instance.
- **List<OrderReceipt>** which holds all the order receipts which are currently in the MoneyRegister.
- The **MoneyRegister** object.

5.6 PROGRAM EXECUTION

The BookStoreRunner class is tasked with running the simulation. When started, it should accept as argument (command line argument) the name of the json input file to read, and the names of the four output files- the first file is the output file for the customers HashMap, the second is for the books HashMap object, the third is for the list of order receipts, and the fourth is for the MoneyRegister object.

The BookStoreRunner should read the input file (using Gson parsing). Next it should create the Inventory object (adding the initial inventory to it), create all the required objects, create and initialize the Micro-Services. When the number of passed ticks equal to duration, all Micro-Services should unregister themselves and terminate gracefully. After all the Micro-Services terminate themselves, the BookStoreRunner should generate all the output files and exit. Important: all the threads in the system should be terminated gracefully.

You must make sure that all the Micro-Services **will not miss** the first TickBroadcast (because they may have not yet been initialized).

5.7 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:

- MessageBus.
- Inventory.
- Future.

You need to submit the unit tests by 28.11.2018 (before the deadline of the assignment).

Here are some instructions. Notice: In Maven the unit tests are placed in src/test/java. These are the steps:

1. Download the interfaces.
2. Extract them.

3. Import Maven Project in Eclipse – choose the folder which contains src and pom.
4. To add a test for class X, right click on the class X, add -> new -> JUNIT Test Case, and place it in src/test/java.
5. Submit all your package, with all the classes.
    a. Make sure you compile with Maven.

5.8 TESTING YOUR ONLINE-BOOK STORE

- You should write input files with json format. We supplied you with one input file which you can use it as example. The input file we supplied you is simple, it does not test all possible cases in the system. 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?
    - Invariants:
        - The number of books sold of some book cannot exceed the available number of books on the inventory.
        - In each tick, the receipts of a customer must reflect his orders till that tick (for example – if a customer wishes to order some book at tick 5, we do not expect to find a receipt for this order with orderTick at tick 4).
        - The amount charged from the credit card of a customer cannot exceed the available money in that credit card.
    - Correctness:
        - If a customer ordered a book which is available on the inventory and there is enough money in his credit card, then his order should have been confirmed.
            - One possible scenario is: there are two customers C1 and C2 which ordered the same book concurrently, and there is only one book available. Customer C1 has enough money to pay, but Customer C2 does not. In this case, customer C1 should take the book.
    - Consistency
        - All the receipts of a customer are available in both the MoneyRegister and the Customer objects.
        - The *amountInInventory* for each book must reflect the actual amount on the inventory.
        - The availableAmount in a credit card must reflect the actual available amount.
        - The earnings of the store reflects all the money gained from the orders.
        - Each confirmed order should have a corresponding receipt for the customer.
    - Round-Robin scheduling: the events are processed by the services in round-robin, e.g., if there are 5 events of OrderBookEvent and 5 services of SellingService, we expect that each SellingService processed one event. The same goes for all the other services and events.
    - Delivery: we will also check that the delivery mission works properly by checking that the threads have slept the required amount for delivery.

5.9 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).

5.10 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 the 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 partners name.
    - Your id and your partners id.
    - Explanation regarding the reason of the extension request.
    - Official certification.

  Request without a compelling reason will not be accepted.

5.11 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".