**QUEUES SIMULATOR**

1. **Main Objective**

The purpose of this project is to design and implement a simulation that aims to analyse queuing based systems for minimizing clients’ waiting time.

The application is made to simulate a queue system, by implementing a multithreading strategy to minimize the waiting time for clients depending on the number of queues available. The application should take its data inputs form the in-text.txt file and to write into another file the output which is represented by the evolution of the queues and of the clients, calculating in the end the average minimum waiting time.

1. **Analysis, assumptions and data modelling**

When we are taking about the analysis of a problem, we mainly talk about an abstract set of operations and properties with the help of which we are able to find some characteristics of the processes. This advantage is done by the object oriented programming; this strategy is called bottom-up design.

Usually we start from the specifications of the project, where we are looking for:

* Noun – which becomes potential classes;
* Verbs- which can play the role of the methods inside the classes.

We should start by explaining the usefulness of the program and the way of processing data:

Let’s assume we want to have 5 queues for waiting customers , 20 customers and a simulation interval of 30 seconds, so that all customers will be served, regardless of their service time – in the limits of the minimum and maximum of their service time-.

All the data will be introduced in the in-text.txt file, which will generate a list of customers( each with their own random arrival and service time) and initialize the queues, by instantiating them and starting their corresponding threads, since each queue has its own thread which must be run in order to be able to serve their customers.

After starting the simulation of the app, the main thread will be started which is used to keep track on the current simulation time and process the customers by associating them to the best queue that is available.

In the out-text.txt file the status of each queue is written at each second, also associating them their corresponding customers with the arrival time and the service time left to be done. When the time interval is over, the program will print the conclusion of the simulation which is represented by the minimum waiting time for a customer.

*Assumptions:*

We will further assume that the input will respect the logical requirements such as minimum is greater than maximum and that the simulation time will be enough for all customers to be served.

1. **Design**

The class that controls the data flow is called Store which can be viewed as the controller of our application.

The classes that do the computational process are called: MyData, Queues, Customers and app which is basically only an instantion for the main method, many of them extending Runnable.

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

void run()

When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

The general contract of the method run is that it may take any action whatsoever.

Now, we will present briefly the main idea of the classes:

The Customer class has 2 attributes which represents the serving time or the processing time and the arrival time, which will be generated randomly.

The Queue class is using an Atomic Integer for the waiting time in order to provide a the thread safety. The Atomic Integer also has some predefined methods such as get() which enables us to get the int value of the variable, addAndGet(int ) which atomically adds the given value to the current value, getAndSet( int) which atomically sets to the given value and returns the old value or getAndUpdate( Unary Operator) which atomically updates the current value with the results of applying the given function, returning the previous value.   
*Atomicity* deals with which actions and sets of actions have indivisible effects. This is the aspect of concurrency most familiar to programmers: it is usually thought of in terms of mutual exclusion.

In the Queue class we also have implement a Linked Blocking Queue; The LinkedBlockingQueue class implements the Blocking Queue interface.

The Linked Blocking Queue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX\_VALUE is used as the upper bound.

The Linked Blocking Queue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a Linked Blocking Queue:

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

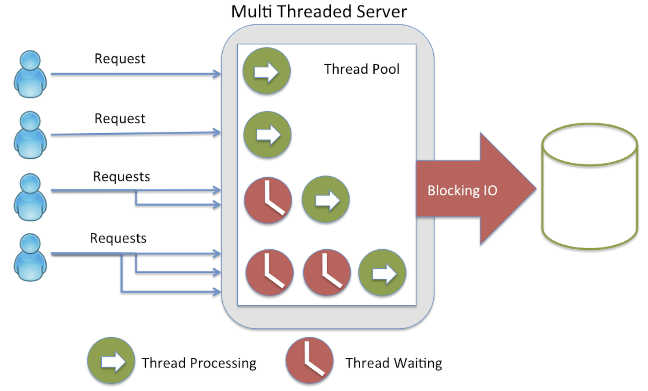
bounded.put("Value");

String value = bounded.take();

The most important class is called Store and its main function is of being the main controller of the application.

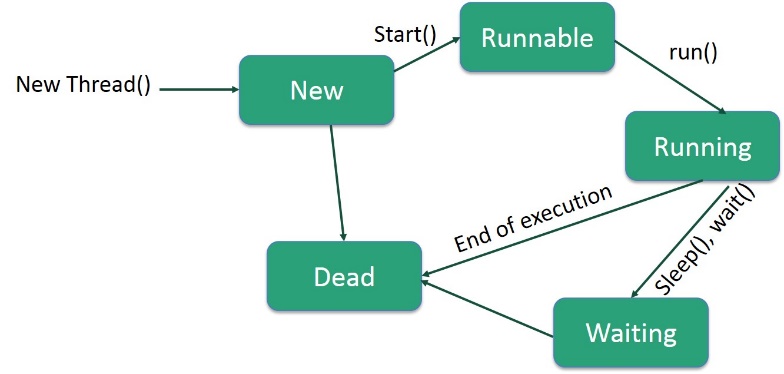
Here are instantiated the files form where the data is read and is written to, moreover here is created the list of queues, customers and threads; here is the implementation of multithreading.

One of the most common scenarios where using multiple threads significantly improve an application's performance is a client-server application. A single threaded application means only one client can connect to the server at a time, but a multi-threaded server means multiple clients can connect to the server at same time. This means next client don't have to wait until your application finish processing request of the previous client.  
  
Like in the following example, you can see that multiple requests are processing by our multi-threaded server at same time.



Threading is not free, it comes with its own challenges. You can only maximize throughput of your application up to certain extent, once numbering of the thread increases up to a certain threshold, they will start competing for CPU and context switching will occur. Context switching means one thread which is using CPU is suspended and CPU is allocated to another thread for execution. When this happens, thread generally loses all of their cached data. If that thread resumes on another core then it has to build its cache from the start.  
  
Threading also introduces a special set of a problem known as multi-threading issue e.g. deadlock, livelock, memory inconsistency error, race conditions, and starvation. It's also very difficult to test a Java program which involves multiple threads. You cannot predict the order of execution in case of multi-threading and without any synchronization.

The life cycle of a thread:



Following are the stages of the life cycle −

* **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
* **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
* **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
* **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
* **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

**Thread Priorities**

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

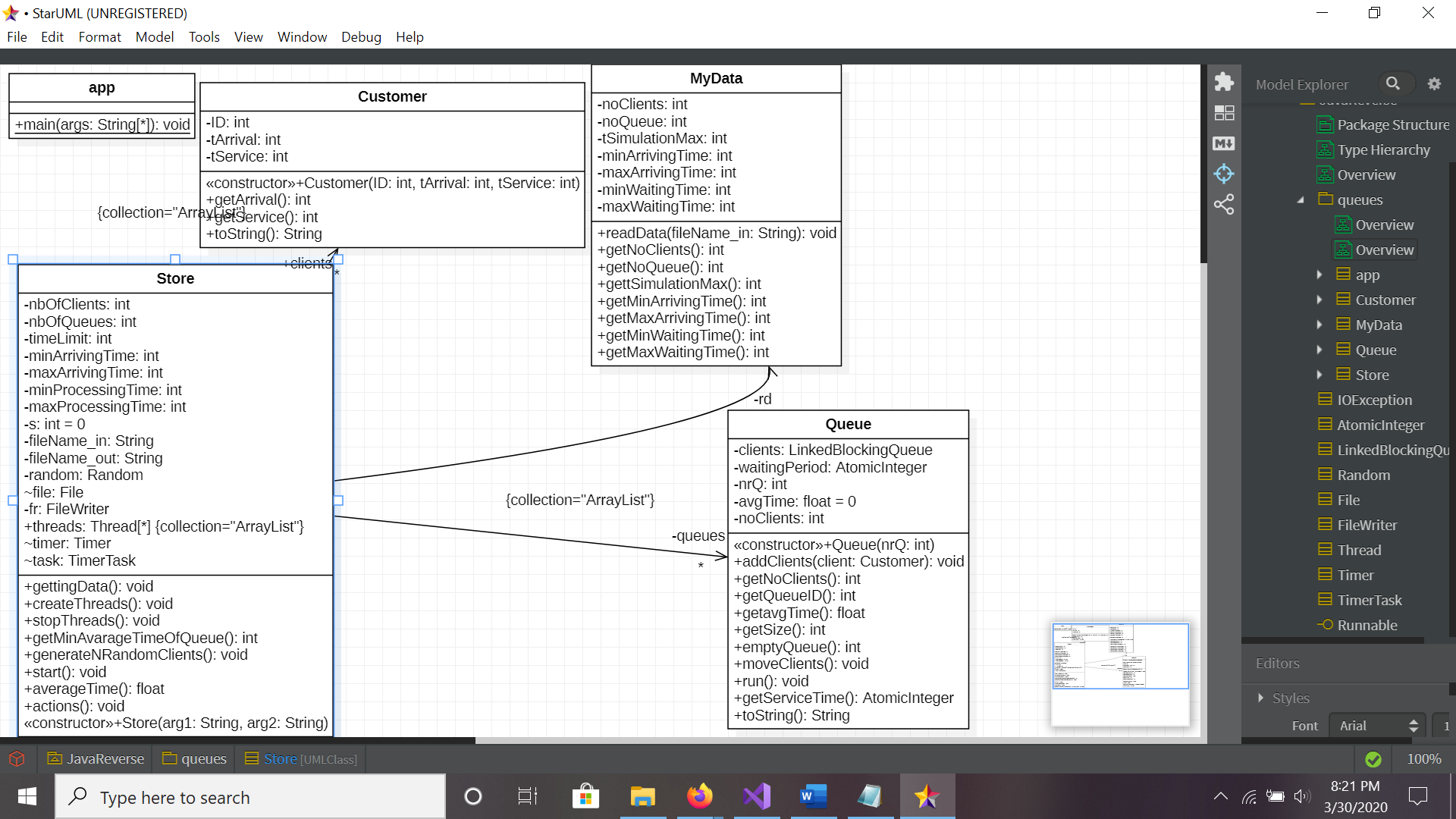
So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block.

**Thread Deadlock**

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

Now, we will present the UML Diagram:



1. **Implementation**

Further, I will briefly describe the most essential methods presented in the application:

Queue.java :

1. Attributes:

* LinkedBlockingQueue<Customer> clients ;
* AtomicInteger waitingPeriod;
* Int nrQ;
* Float AvgTime;

1. Methods:

* Run() – this method overrides the run() method of the Runnable interface
* addClient() – add a client to the list of clients
* moveClient() – it gets a client out of the queue

Store.java:

1. Attributes:

* int nbOfClients;
* int nbOfQueues;
* int timeLimit;
* int minArrivingTime;
* int maxArrivingTime;
* int minProcessingTime;
* int maxProcessingTime;
* private ArrayList<Queue> queues;
* public ArrayList<Thread> threads;
* public ArrayList<Customer> clients;

1. Constructor:

* Takes as parameters the 2 files; an in and an out file for reading and writing
* Instantiate all the methods needed in order to run the application

1. Methods:

* generateNRandomClients() – generate random times for arriving and service of each client
* run() -this method overrides the run() method of the Runnable interface

-let the application to run as long as the time indicated by the timer is smaller or equal to the time limit of the simulation given as a parameter

* actions() – generate the “ action” of the application printing also the states of the queues the simulation

1. **Results:**

The final application is able to calculate the average time that a client has to wait for the queue, using multithreading and exception handelers.

1. **Conclusion:**

The main purpose of this assignment was to get familiarized with asynchronous programming and multithreading. This is my first application where I had to deal with multithreading in order to empower the running time of the program.

Doing this assignment I have also learnt how to generate a .jar file and I was able to understand the need of generating it. In simple words, a JAR file is a file that contains compressed version of .class files, which were basically designed in order to let the developer to distribute a version of his/ her software, distributing a single file and not a directory structure filled with class files.

1. **Possible improvements**

Some of the improvements that can be brought to the application are building a Strategy pattern, so that more statistics can be made based on the given parameters, also this application could be done using the strategy pattern, but taking into account the simplicity of the methods and it’s reduce number, it does not have a great impact .

1. **Bibliography**

<https://javarevisited.blogspot.com/2017/03/why-we-use-threads-in-java.html>

<http://tutorials.jenkov.com/java-util-concurrent/linkedblockingqueue.html>

<https://www.geeksforgeeks.org/working-with-jar-and-manifest-files-in-java/?ref=rp>

https://stackoverflow.com/questions/3140731/isalive-method-of-java-thread-is-not-working-properly

<https://www.studytonight.com/java/joining-a-thread.php>

http://www.periodicooficial.oaxaca.gob.mx/files/2011/05/ EXT02-2011-05-19.pdf