Lab 1 - "Non-cooperative" multithreading

Supermarket inventory:
There are several types of products, each having a known, constant, unit price. In the begining, we know the quantity of each product.
We must keep track of the quantity of each product, the amount of money (initially zero), and the list of bills, corresponding to sales. Each bill is a list of items and quantities sold in a single operation, and their total price.
We have sale operations running concurrently, on several threads. Each sale decreases the amounts of available products (corresponding to the sold items), increases the amount of money, and adds a bill to a record of all sales.
From time to time, as well as at the end, an inventory check operation shall be run. It shall check that all the sold products and all the money are justified by the recorded bills.

The following classes are present:

Product
- It has a name and a price
- Methods getName() and getPrice() which return the name and the price

Bill
- It is a concurrent hash map of product and and integer that is the quantity. It is a hash map because multiple threads can operate on a single object without any complications
- In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate
- There are 2 methods: getTotalValue() and getTotalQuantity() which return the totalValue and the totalQuantity

Sale
- implements Runnable interface in order to be used by threads
- we check if there are available items on stock and if there are we sell them
- the thread.sleep(500) method is used for the other threads to enter the synchronized block

SupermarketInventory
- we have a logger in which we add the info about the threads
- a map between product and the quantity which is Integer
- a list of bills which is synchronized
- a revenue which is atomicLong because it is thread safe
- an initial quantity which is 0
- addProduct() function which is used to add random products
- sellProducts() function in which we first check if there are available items on stock, after that we get a random product, and lock it to perform operations on it. We first check the old quantity of the product, then create randomly a new quantity to be sold. The new quantity is the difference between the old quantity and the sold one
- we also use the synchronized method to calculate the money amount and add a bill to the bill list

- ❖ we log the product sold in the current thread and the quantity
- ❖ in the getAvailableProducts() we have a filter in which we get the remaining products
- ❖ getSoldProductsValue() we have an atomic value, we lock the bill list and get the price for the current bill list
- ❖ same for getSoldProductsQuantity()
- ❖ we have an inventory check in which we use the getSoldProductsValue() to see if the threads are working correctly
- ❖ in the final inventory check we see if the money and the quantity are correct

InventoryCheck

- ❖ in this class we use the methods from the SupermarketInventory class: runInventoryCheck and runFinalInventoryCheck
- ❖ it is also a runnable in order to be used by threads

Main

- ❖ we use the SupermarketInventory class and InventoryCheck class
- ❖ we use a method to create random products with a random price
- ❖ addProductsToSupermarket to add each product with a random minQuantity < quantity < maxQuantity
- ❖ runInventoryCheck() in which a thread is started to check
- ❖ runClients() in which we have a number of clients (number of threads) and each of them buys random products until there are none left

Locking mechanism

```
synchronized (product) {
    int oldQuantity = productsQuantityMap.get(product);
    if (oldQuantity == 0) {
        return;
    }
    // sell a random quantity between 1 and 20, but <= the avalable stock
    int soldQuantity = Math.min(random.nextInt( bound: 20) + 1, oldQuantity);
    int newQuantity = oldQuantity - soldQuantity;
    productsQuantityMap.put(product, newQuantity);
    synchronized (revenue) {
        revenue.set(revenue.get() + (long) product.getPrice() * soldQuantity);
    }
    synchronized (billsList) {
        billsList.add(new Bill(product, soldQuantity));
    }
    StringBuilder productsLog = new StringBuilder();
    productsLog.append(Thread.currentThread().getName()).append(" Client buying ").append(soldQuantity).append("x ").append(p
    LOG.info(productsLog);
}
```

- ❖ the synchronized method is used
- ❖ it is used to lock an object for any shared resource
- ❖ when a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Time

No of products	5	10	50	75
No of clients	**100**	**250**	**400**	**1000**
Time	**3.008 seconds**	**3.072 seconds**	**9.088 seconds**	**5.184 seconds**

Windows specifications

Edition	Windows 10 Pro
Version	20H2
Installed on	3/24/2021
OS build	19042.1288
Experience	Windows Feature Experience Pack 120.2212.3920.0

Device specifications

Device name	DESKTOP-8TAVK7Q
Processor	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
Installed RAM	8.00 GB
Device ID	5AFAABCC-88DE-4650-AF16-060A05CAD1FD
Product ID	00330-80952-43821-AA158
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display