Assignment Supermarket

Object Orientation

Spring 2020

There is no resit for this assignment, because there is not enough time until the exam.

1 Learning Goals

After completing this assignment, you should be able to:

- Synchronize threads
- Let threads communicate with each other with conditions and `await` and `signal`
- Prevent deadlocks
- Stop threads safely

2 Shopping at SuperFast

In this exercise, you are going to write a Java program that simulates a supermarket. Customers visit the supermarket in order to purchase a number of items. When a customer has collected all their items they go to a random register. Each register has two belts, one incoming before scanning and one outgoing after scanning. The customer waits their turn at the register and then places all items on the incoming belt. The cashier grabs each item from the incoming belt, scans it and puts it onto the outgoing belt. The customer then picks up their items from the outgoing belt and takes them home.

You do not have to write the whole program yourself, we will provide a large part of the implementation as a start project. You should use threads and synchronization for your implementation. You should use Callables and an Executor instead of threads directly. Customers and cashiers should both be modelled as Callables. The conveyor belts function as shared objects which, by using the right synchronization primitives, will ensure that the entire system functions properly.

The class diagram in fig. 1 shows the classes and their relationship.

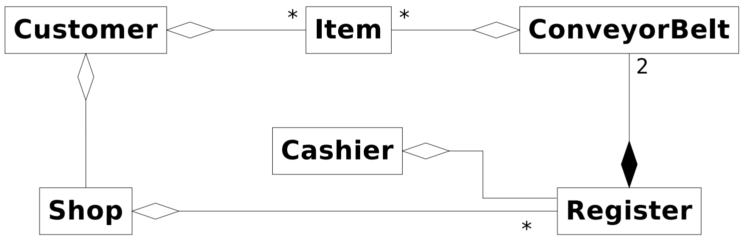

Figure 1: Class diagram for the supermarket.

The cashier and the customer are in a *producer-consumer relationship*. First, the customer serves as a producer by placing the items on the belt, which are then consumed by the cashier. The roles are reversed in the next step. After scanning the items the cashier produces the items, which are consumed by the customer. The belts are used as buffers during the transfer.

2.1 The Class ConveyorBelt

You should make this class suitable to represent the conveyor belts. The belts should have *first-in-first-out* behavior, which means that the item placed first on the belt will also be removed first. The number of items on the belt is limited, so you are dealing with a limited queue.

The constructor has a parameter `capacity`, the maximum amount of elements in the queue. The elements themselves are stored inside an array called `elements`.

The attributes `count`, `begin` and `end` keep track of the actual amount of elements inside the array, the position of the element that was added first and the position at which the next new element needs to be added. The method `add` can be used to add a new element to the waiting queue, the method `grab` deletes the oldest element from the list.

The class ConveyorBelt in the start project can only store integers, but you want to store any data type. You should turn ConveyorBelt into a generic class.

2.2 The Class Item

This class is used to represent the items present in the store. Each item can be identified by a unique number, called `barcode`. As for now, you will not use that number. The class does not have any other methods and does not need to be modified further.

2.3 The Class Store

This class is complete already, you do not need to add anything to it.

It does not explicitly keep track of the items since it is not necessary for the simulation. The store has a fixed amount of registers which are stored in a list, which is initialized in the constructor.

The method `open` creates a cashier thread for every register.

The method `getItems` can be used by customers to grab a given number of items.

The method `claimRegister` returns a register with a number `registerNr`. The customer can use this method to choose a cash register where they will pay for their items.

2.4 The Class Register

You have to implement most of the `Register` class. A register has two conveyor belts, one for the customer to place their shopping items on, and the other for the cashier to put down the items they just scanned. The first one is called *belt* and the and the other one *bin*. You have to implement two methods for each of the two belts that allows to add and remove items.

Customers should be able to use `claim` and `free` to exclusively claim a register.

2.5 The Class Customer

To be able to create a thread for a customer, it implements the `Callable` interface. This means that this class needs to implement the method `call`. This function should put all items on the belt, take all items from the bin, and return the number of items a customer has bought. All the remaining parts of this class are given in the code.

Customers have a reference to the store, and keep track of the amount of items they want to buy. The latter is determined in the constructor using a random generator. The constructor has the store as its only parameter.

2.6 The Class Cashier

This class implements the `Callable` interface as well. Cashiers have a reference to their register.

2.7 Synchronization

Finally, you should add synchronization to the class ConveyorBelt. You have to make sure that the belts are being properly synchronized, and also implement a waiting mechanism for processes that want to either put items into a full queue or take items from an empty queue. You should use the synchronization primitives `Lock`, `Condition`, `await` and `signalAll` to manage and update waiting processes.

3 Your Tasks

1. Turn ConveyorBelt into a generic class.
2. Add synchronization to ConveyorBelt such that:
   - All accesses to the elements are synchronized.
   - When the put method is called while elements is full, the calling thread should wait until there is space.
   - When the get method is called while elements is empty, the calling thread should wait until there are items.
   - Hint: You need one Lock and two Conditions.
3. Implement the class Register.
   - Add attributes, create a constructor and implement the class methods.
   - To make sure that customers products are not mixed up on the belt, a customer must claim a register and release it when done. Create the corresponding methods.
   - Hint: You need one Lock.
4. Implement the class Customer. The `call` method should behave as follows.
   (a) Collect all the items he wants to buy from the store.
   (b) Choose a register. Choosing the register should be done with a random number.
   (c) Claim the register.
   (d) Place all items on the belt.
   (e) Grab all items from the bin, while counting them. This is the number of items bought.
   (f) Free the register.

   (g) Return the number of items bought.

   To indicate that all items from their shopping cart are placed on the belt, the customer puts a `null` onto the belt. This represents the little separating log that is used in supermarkets to indicate where the items of the next customer begin.

5. Implement the class Cashier. Implement the `call` method in which cashiers help their customers. To keep it simple, you should not include a function to signal the cashier when their work is over. This means that their task is an endless loop in which they transfer items from the belt to the bin of their register..
6. When all customers are done, the `main` function should print the total number of items sold. Use the list of futures to sum all numbers returned by the customers. Hint: invokeAll blocks until all callables are done.
7. When all customers are done, the cashiers and the executor should shut down.
8. Hint: Only the classes ConveyorBelt and Register should contain synchronization code.

4 Submit Your Project

To submit your project, follow these steps.

1. Use the **project export** feature of NetBeans to create a zip file of your entire project: File → Export Project → To ZIP.
2. **Submit this zip file on Brightspace**. Do not submit individual Java files. Do not submit any other archive format. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.