TDT4186 Mandatory exercise: SushiBar

by Kristoffer Nyvoll

**Note**: I’ve included this .docx-file in case this is what you prefer - the formatting is far superior if you check out the “answers.md” - file

If you don't have a Markdown-editor, just view the file here: [Hackmd.io](https://hackmd.io/@4UXtod5YQ2OWRBOzgVH2hA/HkVxTdjUI)

### **Explanation of my implementation**

The sushibar system is an example of the **consumer/producer** problem in Java. This problem is a classic example of a multi-process syncronization problem. The essence of the problem is that two processes; the consumer (waitress) and the producer (door) share the same resource (waitingarea). We have to make sure that the consumer won't try to remove data from an empty buffer and that the producer won't try to add data to a full buffer.

The solution to this, which is the foundation upon which i implemented this code, is as follows:

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Source: [Link](https://www.geeksforgeeks.org/producer-consumer-solution-using-threads-java/).

Here is an example of how i implemented this:

// Essence of the producer/consumer model: Producer has to wait if the shared resource is full

if (waitingArea.isFull()) {

while (waitingArea.isFull()) {

try {

synchronized(waitingArea) {

waitingArea.wait(); // <- This is the important bit

}

} catch (InterruptedException e) {

e.printStackTrace();

}

To put it clearly:

1. wait() is used in both Door.java and Waitress.java and this is interrupted when two separate calls to methods in the WaitingArea.java is invoked.
2. If the WaitingArea is full, the door waits
3. When a customer is recovered from the WaitingArea, the Door wakes up and lets a new customer in.
4. Similarly, the Waitress waits if the WaitnigArea is empty, and wakes up if a new customer comes through the Door.

### **Answers**

**a)**

*wait()* - this method causes the current (this) thread to wait until notify() or notifyAll() is invoked. No parameters or return value. Using the method releases ownership of the thread until it is "woken up" by another call to the notify/notifyAll method.

*notify()* - "wakes up" a single thread. If there are multiple threads waiting for this object, the selected is chosen arbitrarily.

*notifyAll()* - Unlike notify(), this method wakes up all the waiting threads - which is the difference between the two methods.

**b)** - Which variables are shared variables and what is your solution to manage them?

We have couple of shared variables - for example these from SushiBar.java

// Variables related to statistics.

public static SynchronizedInteger servedOrders;

public static SynchronizedInteger takeawayOrders;

public static SynchronizedInteger totalOrders;

And "customers" from WaitingArea.java. This last one must be managed individually, and was solved by adding the "synchronized"-flag to every interaction involving the "customers" variable.

"SynchronizedInteger" solves this problem for us, so that does not need any additional implementation.

**c)** - Which method or thread will report the final statistics and how will it recognize the proper time for writing these statistics?

try {

producerThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

​

SushiBar.write("Total number of orders: " + totalOrders.get());

SushiBar.write("Total number of takeaway: " + takeawayOrders.get());

SushiBar.write("Total number of orders eaten at the bar: " + servedOrders.get());

SushiBar.java writes out the stats to the log.txt file if the "try" block above is succesful. This happens when all the other threads are done (terminated). Having this at the very end of the main() method in addition to having a try-catch-block for extra safety ensures that this only occurs when all the customers are done