NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

TDT4186

Operativsystemer

Practical Assignment: SushiBar

Ådne Karstad

19. mars 2020

1 Description of Implementation

1.1 SushiBar.java

Is the where the main method is run and where all the other classes are instantiated. Looking at figure 1 you see that the Sushibar is the first object being insantiated, followed by the clock which counts down the time the Sushi bar is kept open. The clock is also the first Thread that begins. Then the waiting area is instantiated as a normal object, followed by a door thread and seven individual waitress threads. I have then set the code to wait for all the waitress threads to die before continuing, making sure that the statistics in the end for the main method is printed out after the Sushi bar is closed and all customers have been served.

```
39
40          // TODO: initialize the bar and start the different threads.
41          write( str: "Instantiate the Sushi Bar");
42          SushiBar sushiBar = new SushiBar();
43
44          // Instantiate the clock as a thread to open the restaurant
45          write( str: "Instantiate Clock!");
46          Thread clock = new Clock(duration);
47          write( str: "Clock Instantiated!");
48
49          // Instantiate the threading classes
50          write( str: "Instantiating waiting area");
51          WaitingArea waitingArea = new WaitingArea(waitingAreaCapacity);
52          // Begin the thread taking in the runnable door
53          write( str: "Instantiating Door");
54          Thread doorThread = new Thread(new Door(waitingArea),  name: "Door-thread");
55          doorThread.start();
56          write( str: "Door: " + doorThread.getName() + " has come alive!");
57
58          // Begin the waitresses
59          write( str: "Instantiating Waitresses");
60          List<Thread> waitressList = new ArrayList<>();
61          for(int i = 0; i < waitressCount; i++) {
62              Thread currentThread = new Thread(new Waitress(waitingArea),  name: "Waitress " + (i+1));
63              currentThread.start();
64              waitressList.add(currentThread);
65              write( str: "Waitress: " + currentThread.getName() + " has come alive!");
66          }
67
68          for (Thread waitress : waitressList)
69              waitress.join();
70
71          SushiBar.write( str: "***** STATISTICS *****");
72          SushiBar.write( str: "TOTAL NUMBER OF ORDERS: " + SushiBar.totalOrders.get());
73          SushiBar.write( str: "TOTAL NUMBER OF TAKEAWAY ORDERS: " + SushiBar.takeawayOrders.get());
74          SushiBar.write( str: "TOTAL NUMBER OF SERVED ORDERS: " + SushiBar.servedOrders.get());
75          SushiBar.write( str: "TOTAL NUMBER OF CUSTOMERS SERVED: " + SushiBar.numberOfCustomers);
76          SushiBar.write( str: "***** END OF STATISTICS *****");
77
78      }
```

Figure 1: SushiBar

1.2 WaitingArea.java

This class is an orchestrator which keeps count of how many customers are inside the waiting area and has the two synchronized methods enter() and next(). The two methods, respectively, are producer and consumer methods. The synchronized tag is provided to ensure that only one Thread may access the given method (or resource) at a time.

1.2.1 enter(Customer customer)

It is the Door thread that will be using the enter method and it's sole purpose is to produce the customers. I have added a check if the customerQueue is full (meaning the waiting area is full), to ensure that the Door is not overproducing customers. If it is currently full then the Door thread will be put to waiting, and will have to receive a notify to wake up again. Similarly, when the door actually produces customers it wants to notify the waitresses to ensure that they are kept running when there are customers in the waiting area.

1.2.2 next()

The next method is used by the waitress threads and it's sole purpose is to consume the customers. The customerQueue is an actual queue data structure, so that the poll method invoked on the queue takes the head of the queue and returns it. If the queue is empty, the poll method will return null instead. Similar to the enter method a waitress wants to notify the door that customers are being consumed, so that the door is kept running to produce more costumers.

```
23      /**
24       * This method should put the customer into the waitingArea
25       *
26       * @param customer A customer created by Door, trying to enter the waiting area
27       */
28      public synchronized void enter(Customer customer) throws InterruptedException {
29          while (isFull()) {
30              SushiBar.write( str: "The waiting area is currently full and " +  Thread.currentThread().getName() + " will have to wait be
31              wait();
32          }
33          this.customerQueue.add(customer);
34          SushiBar.write( str: "Customer " + customer.getCustomerID() + " has entered the waiting area and is now waiting.");
35          notifyAll();
36      }
37
38      /**
39       * Consumer method.
40       * @return The customer that is first in line.
41       */
42      public synchronized Customer next() throws InterruptedException {
43          Customer customer = customerQueue.poll();
44          notifyAll();
45          return customer;
46      }
47
48      /**
49       * @return True if the customer queue is full, false otherwise
50       */
51      public boolean isFull() {
52          return this.size <= customerQueue.size();
53      }
54
55      /**
56       * @return True if the customer queue is empty, false otherwise
57       */
58      public boolean isEmpty() {
59          return this.customerQueue.size() == 0;
60      }
61
62  }
```

Figure 2: Waiting Area

1.3 Door

The door wants to simulate an actual flow of customers to the restaurant, thus after actually producing a customer the Thread will be put to sleep for a random length of time.

```java
/**
 * This method will run when the door thread is created (and started).
 * The method should create customers at random intervals and try to put them in the waiting area.
 */
@Override
public void run() {
    while (SushiBar.isOpen) {
        try {
            waitingArea.enter(new Customer());
            Thread.sleep((int)(Math.random()*SushiBar.doorWait));
        } catch (InterruptedException e) {}
    }
}
}

```

Figure 3: Door

1.4 Waitress

The waitress wants to keep running even when the Sushi bar is closed, but only as long as the waiting area is not empty. Hence the while(Sushibar.isOpen || !waitingArea.isEmpty()). I've added a check if the customer consumed by the waitingArea.next is null because poll may return null if the queue is empty, and will simply just try anew if it is null. To simulate a real waitress the Thread is put to sleep for a given waitressWait time before taking the order, and then again customerWait time after the customer has made his or her order.

```java
/**
 * This is the code that will run when a new thread is
 * created for this instance
 */
@Override
public void run() {
    // TODO: Implement required functionality.
    while(SushiBar.isOpen || !waitingArea.isEmpty()) {
        // Run this as long as the bar is open or there is someone still in the waiting area.
        try {
            Customer customer = waitingArea.next();

            // If there were no customers in the waiting area then customer will be null
            if (customer != null) {
                SushiBar.write( str: Thread.currentThread().getName() + " is waiting some time before taking the order.");
                Thread.currentThread().sleep(SushiBar.waitressWait);
                SushiBar.write( str: Thread.currentThread().getName() + " is now taking the order for customer: " + customer.getCustome
                customer.order();
                Thread.currentThread().sleep(SushiBar.customerWait);
            }
        } catch (InterruptedException e) {
            SushiBar.write( str: "The Queue is empty " + Thread.currentThread().getName() + " is waiting for some more customers.");
        }
    }


```

Figure 4: Waitress

1.5 Customer

The customer id is simply utilizing the SyncrhonizedInteger class provided in the exercise to increment a simple integer id for each customer. The order method allows the customer to order a random amount of orders in the closed interval of [0, Sushibar.maxOrder]. The number of served orders is then a random amount in the closed interval [0, numberOfOrders], and the number of take away orders is the difference between the number of orders and the number of served orders.

```
/**
 * Here you should implement the functionality for ordering food
 * as described in the assignment.
 */
public void order() {
    // TODO: Implement required functionality.
    int numberOfOrders = (int) (Math.random() * SushiBar.maxOrder);
    int numberOfServedOrders = (int) (Math.random() * numberOfOrders);
    int numberOfTakeAwayOrders = numberOfOrders - numberOfServedOrders;

    SushiBar.write( str: "Customer " + getCustomerID() + " is ordering \nTakeaway: " + numberOfTakeAwayOrders + "\nServings: " + number
    SushiBar.servedOrders.add(numberOfServedOrders);
    SushiBar.takeawayOrders.add(numberOfTakeAwayOrders);
    SushiBar.totalOrders.add(numberOfOrders);
}
```

Figure 5: Order Method

2 Exercise Questions

1. What are the functionality of wait(), notify() and notifyAll(), and what is the difference between notify() and notifyAll()?
2. Which variables are shared variables and what is your solution to manage them?
3. Which method or thread will report the final statistics and how will it recognize the proper time for writing these statistics?

2.1 Question 1

2.1.1 wait()

Wait puts the current Thread into blocked and will require a notify() or notifyAll() to be put to ready again.

2.1.2 notify()

Will signal a single Blocked Thread to be ready and have the possibility to be running again.

2.1.3 notifyAll()

Will signal all blocked Threads (related to the given resource) to be ready, and then they will have to "fight" for the resource they were blocked from.

2.1.4 Difference between notify() and notifyAll()

Whereas notifyAll wakes up all the Threads, the notify method only wakes up a single thread at a time. This may lead to a problem if the single thread woken up by notify is not able to take hold of the resource but instead is put back to blocked.

2.2 Question 2

A share variable is the costumerQueue which hold references to the customer objects. The challange with sharing the queue is that when several workers (both producer(s) and consumer(s)) want access to the queue it may overwrite each other since the Threads are working in parallel. A way to avoid the queue to overflow, or "underflow?", is to make sure that the Door (producer) and Waitress (consumer) wait if some condition is satisfied. The Door should wait if the queue is full, and similarly the waitress should wait if the queue is empty. Another method to hinder several workers to access a resource is to tag it with a synchronized keyword, this may be done on any Data Structure or method, and makes sure that only one Thread may work on the resource at a time.

2.3 Question 3

I believe the main method actually does the writing of the statistics in my code, but it waits for the all the waitress Threads to be complete before writing it. Thus the SushiBar is the actual writer of the statistics.