TDT4186 - Assignment 1

Erik Liodden (eriklio)

March 2018

Explanation of implementation

The sushi bar problem given in the assignment is implemented using the *consumer/producer model*. The *door* is a producer (generates more customers and put them in the *waitingArea*). The *waitress* is a consumer (takes customers out of the *waitingArea*). The producer (the *door*) and the consumers (the *waitress*) run on separate threads. The *waitingArea* is a shared variable between these threads, and needs to protect itself from simultaneous access from different threads.

By adding `synchronized` to methods to the object of a shared variable, only one thread can access any of the synchronized methods of the instance at a given time. This ensures that the instance of the shared object cannot be modified at the same time.

The threads themselves that access a shared variable can communicate with each other through `wait()`, `notify()` and `notifyAll()`. When the desired resource is not available for a thread, the thread will `wait()` and will not take CPU time. Other threads that access the same instance of an object can call `notify()` or `notifyAll()` when they have modified the resource such that other waiting threads can be interrupted and possibly resume their execution. `notify()` will interrupt a random waiting thread while `notifyAll()` will interrupt all. I have used `notifyAll()` in my implementation. An example from `WaitingArea.java` is shown below.

```
1  /**
2  * @return The customer that is first in line.
3  */
4  public synchronized Customer next() {
5          while (queue.size() == 0) {
6                  try {
7                          wait();
8                  } catch (InterruptedException e) {
9                          e.printStackTrace();
10                 }
11         }
12         notifyAll();
13         return queue.poll();
14 }
```

The thread accessing `waitingArea.next()` will wait until there are people in the queue before returning the next customer in the queue and notify all other waiting threads of the change.

The `SushiBar.java` runs the main thread and is responsible to create, start and join all other threads. The `main()` in `SushiBar.java` creates the producer (*Door*), the consumers (*Waitress*) and a timer (*Clock*). It then starts all threads. The consumers and producer will run as long as the *isOpen* flag is set. The timer set this value to *false* when the program has run as long as specified by *duration*. When the *isOpen* flag is set to *false*, the producer will stop generating new customers, but the consumers will continue until there are no more people in the waiting queue. The threads themselves will finish normally and is then joined by the `main()` method in `SushiBar.java`. When the all the threads are joined, the `SushiBar.java` class then writes the final statistics to the log (`log.txt`).

Source Code

The source code for the implementation of Assignment 1 can be found in `src.zip`. The solution is implemented in java, using Java 1.8.