Exercise 13: Threads and the producer-consumer problem

A restaurant has n chefs and m waiters working there. The chefs cook at a furious pace and put the prepared meals in a serving hatch. The waiters stand and wait in front of the serving hatch. Whenever a meal is ready, a waiter takes it and brings it to the guest.

The serving hatch has a capacity of k meals, i.e. as soon as there are k or more meals in the serving hatch, a chef must wait with their already cooked meal until the next waiter has taken a meal away, and the chef can then put their meal in the serving hatch.

Task 1: Implement simple (non-synchronised) serving hatch

Implement the given ServingHatch interface:

```
public interface ServingHatch<T> {
    T get();   // take something out of the serving hatch
    void put(T o);  // put something in the serving hatch
}
```

Note:

- A serving hatch should have a maximum number of "spaces" for meals.
- Use the java.util.Queue interface and (for example) the LinkedList class from the Java library; do not use classes from the java.util.concurrent package.

Task 2: Synchronise the serving hatch

The serving hatch will later be used by several chefs and waiters at the same time. So make sure that the synchronisation is correct: The queue may only be used by one thread at a time. Or in other words: in both `put` and `get`, the access to the queue must take place in a critical section (`synchronized (...)`). The following also applies: If the queue is full, then `put` must wait (wait()) until there is space again; if the queue is empty, then `get` must wait until something is there again. If we are in a critical section, then wait and/or notifyAll can be called on the "key object" in order to wait or to wake up other threads respectively. Normally, if something is added or removed, all other threads are always woken up (someone could be waiting...).

Reminder:

```
class Class {
    synchronized void method1() {
        wait();  // sleep until another on this object calls notifyAll
        notifyAll();  // wake other threads that are currently waiting for this
object
    }
    // or equivalent to key object
    void method2() {
        synchronized (this) {
            this.wait();
            this.notifyAll();
        }
    }
    // or to another key object
    Object key = new Object();
    void method3() {
        synchronized (key) {
            key.wait();
            key.notifyAll();
        }
    }
}
```

It is important to remember which object is used to save (this or another?), because accordingly either wait or notifyAll must be called for this object.

Task 3: Waiter and chef

Implement a chef as a `Runnable` that has a name (`String`) and (successively) produces a specified number of meals and places them in the `ServingHatch<Meals>`. Similarly, implement a waiter that understandably does not cook, but instead completes a certain maximum number of waiter activities (taking a meal to the table).

Note:

- From the above text, you should have realised that both chef and waiter each accept a String (name), an int value (number) and a ServingHatch<Meals> in the constructor!
- A chef should wait a certain (random) time before putting a meal in the serving hatch, in order to simulate the cooking time; similarly waiters should wait a bit after taking a meal in order to simulate carrying it to the table. (e.g. `Thread.sleep((int) (Math.random() * 3000))`)
- Document the calls of `put` and `get` by appropriate outputs to `System.out`, e.g. "`Meals <...> {in,out} ServingHatch {put,taken}`", similar to the output example below.

Task 4: Programme

Write a programme called Restaurant (i.e. a `Restaurant` class with a `main` method), in which a serving hatch is first created, and then passed on to chefs and waiters. Vary the number of chefs, waiters and meals as well as the capacity of the serving hatch.

- Can it happen that a queue forms in front of and behind the serving hatch, i.e. both chefs and waiters are waiting in line?
- Must we consider this when implementing the ServingHatch class?
- What should be done to model the ordering processes?

Please note: Start each chef and each waiter in a separate thread. The programme will terminate as soon as all chefs and waiters have reached their respective limit.

A possible output might look like this:

```
Hans put meal 0 in the serving hatch.

Bernd took meal 0 out of the serving hatch.

Albert took meal 0 out of the serving hatch.

Peter put meal 0 in the serving hatch.

Peter put meal 1 in the serving hatch.

Gisela took meal 1 out of the serving hatch.

Hans put meal 1 in the serving hatch.

Bernd took meal 1 out of the serving hatch.

Peter put meal 2 in the serving hatch.

Albert took meal 2 out of the serving hatch.

Hans put meal 2 in the serving hatch.

Gisela took meal 2 out of the serving hatch.

Hans put meal 3 in the serving hatch.

Bernd took meal 3 out of the serving hatch.

Peter put meal 3 in the serving hatch.

Albert took meal 3 out of the serving hatch.

Peter put meal 4 in the serving hatch.

Gisela took meal 4 out of the serving hatch.

Peter put meal 5 in the serving hatch.

Gisela took meal 5 out of the serving hatch.
```