Exercise 2 - Monitor and Wait/Notify

In this exercise you will re-implement the Buffer class from exercise 1 as a monitor and you will write a monitor for a queue application.

Buffer

Implement the *getLine* method below. Note that the counting semaphores have been replaced with attributes used together with the Java signalling mechanism wait/notify.

```
class Buffer {
    int available;      // Number of lines that are available.
    final int size=8;       // The max number of buffered lines.
    String[] buffData;  // The actual buffer.
    int nextToPut;      // Writers index.
    int nextToGet;      // Readers index.

    Buffer() {
        buffData = new String[size];
    }

    synchronized void putLine(String inp) {
        try {
            while (available==size) wait();
        } catch (InterruptedException exc) {
            throw new RTError("Buffer.putLine interrupted: "+exc);
        };
        buffData[nextToPut] = new String(inp);
        if (++nextToPut >= size) nextToPut = 0;
        available++;
        notifyAll(); // Only notify() could wake up another producer.
    }

    synchronized String getLine() {
        // Write the code that implements this method ...
    }
}
```

A note on InterruptedException

When we in a *synchronized* method wait for some condition to become true, that is, when we call *wait*, we are forced to (by the compiler) to take care of an *InterruptedException* by

- having the method declared to throw that exception, or
- catching the exception and completing the method under the actual situation, or
- catching the exception and terminating the program, or
- catching the exception and generate an unchecked exception like ‘divide by zero’, or
- catching the exception and throwing an instance of (a subclass of) *Error*.

Having examined these alternatives we recommend (in most cases) to throw an *Error*. The purpose of the standard *Error* class in Java is to throw unchecked exceptions that the ordinary programmer is not expected to handle, and when a monitor operation fails there is normally nothing else to do than to enter some type of fail-safe or recovery mode (or simply a thread exit in simple applications).

Queue system

A queue system consists of the following hardware:

- A ticket-printing unit with a button that customers press in order to get a numbered queue ticket.
- Buttons at several counters. A clerk pushes the button to announce that he/she is free to handle a customer.
- A display showing counter number and queue number for the next customer to be served.

Develop the control program for the queue-handling system. The thread classes for input and output have already been written. Your task is to develop the monitor class that the threads use. The hardware is encapsulated in an available static class HW. To solve the task you need to answer some questions before implementation:

1. What data is needed in the monitor?
2. Which conditions must be fulfilled to permit each monitor operation to complete?
3. Which events may imply that the conditions have been fulfilled?
4. An impatient clerk could possibly press its button several times. How do you handle that?
5. Write the monitor!

Input to the control system consists of signals from the customer and clerk buttons. Output consists of the numbers to the display. Reading input is blocking which prevents use of polling. Instead, you need one thread for each of the inputs. This agrees with the rule to have one thread for each independent external sequence of events. It is also specified that the display may not be updated more often than each 10 seconds. Therefore, it is most convenient to have a dedicated thread for updating the display. Thus, we have the following threads:

- *CustomerHandler*: Waits for a customer to push the button, calls the monitor method *customerArrived* in the monitor, and then a queue ticket is printed for the customer.
- *ClerkHandler*: Waits for the clerk to push his/her button, then calls the monitor method *clerkFree*. Each instance handles one specified clerk.
- *DisplayHandler*: Calls the monitor method *UpdateDisplay* which returns values for display update. To give customers time to react, there must be at least 10 seconds between each update.

Specification of the monitor (for simplicity of the exercise, exceptions need not be caught):

```
class YourMonitor {
    private int nCounters;
    // Put your attributes here...

    YourMonitor(int n) {
        nCounters = n;
        // Initialize your attributes here...
    }

    /**
     * Return the next queue number in the intervall 0...99.
     * There is never more than 100 customers waiting.
     */
    synchronized int customerArrived() {
        // Implement this method...
    }

    /**
     * Register the clerk at counter id as free. Send a customer if any.
     */
    synchronized void clerkFree(int id) {
        // Implement this method...
    }

    /**
     * Wait for there to be a free clerk and a waiting customer, then
     * return the cueue number of next customer to serve and the counter
     * number of the engaged clerk.
     */
    synchronized DispData getDisplayData() throws InterruptedException {
        // Implement this method...
    }
}
```

Implementation of the threads in the system. Note that most of the application logic is deferred to the monitor, very little is left in the thread code. This is in general a good design principle for embedded systems. Note also how all threads are sharing the same monitor.

```
class CustomerHandler extends Thread {
    YourMonitor mon;

    CustomerHandler(YourMonitor sharedData) {
      mon = sharedData;
    }

    public void run() {
        while (true) {
            HW.waitCustomerButton();
            int qNum = mon.customerArrived();
            HW.printTicket(qNum);
        }
    }
}

class ClerkHandler extends Thread {
    YourMonitor mon;
    int id;

    ClerkHandler(YourMonitor sharedData, int id) {
      mon=sharedData;
      this.id=id;
    }

    public void run() {
        while (true) {
            HW.waitClerkButton(id);
            mon.clerkFree(id);
        }
    }
}

class DispData {
    int ticket;
    int counter;
}

class DisplayHandler extends Thread {
    YourMonitor mon;
    DispData disp;

    DisplayHandler(YourMonitor sharedData) { mon = sharedData; }

    public void run() {
        while (true) {
            try {
                disp = mon.getDisplayData();
                HW.display(disp.ticket, disp.counter);
                sleep(10000);
            } catch (InterruptedException e) { break; }
        }
    }
}
```