Chapter 4 - exercises

4.1 Recursive Locking in Java

Once a thread has acquired the lock on an object by executing a synchronized method, that method may itself call another synchronized method from the same object (directly or indirectly) without having to wait to acquire the lock again. The lock counts how many times it has been acquired by the same thread and does not allow another thread to access the object until there has been an equivalent number of releases. This locking strategy is sometimes termed *recursive* locking since it permits recursive synchronized methods. For example:

```
public synchronized void increment(int n) {
   if (n>0) {
       ++value;
       increment(n-1);
   } else return;
}
```

This is a rather unlikely recursive version of a method which increments `value` by `n`. If locking in Java was not recursive, it would cause a calling thread to block resulting in a deadlock.

Given the following declarations:

```
const N = 3
range P = 1..2  //thread identities
range C = 0..N  //counter range for lock
```

Model a Java recursive lock as the *FSP* process `RECURSIVE_LOCK` with the alphabet `{acquire[p:P],release[p:P]}`. `acquire[p]` acquires the lock for thread `p`.

Chapter 5 - exercises

5.1 A single slot buffer may be modeled by:

```
ONEBUF = (put->get->ONEBUF).
```

Program a Java class `OneBuf` that implements this one slot buffer as a monitor.

5.2 Replace the condition synchronization in your implementation of the one slot buffer by using semaphores. Given that Java defines assignment to scalar types (with the exception of long and double) and references types to be atomic, does your revised implementation require the use of the monitor's mutual exclusion lock?

5.3 In the museum example (assessed coursework), identify which of the processes, `EAST`, `WEST`, `CONTROL` and `DIRECTOR`, should be threads and which should be monitors. Provide an implementation of the monitor(s).

5.4 FSP allows multiple processes to synchronize on a single action. A set of processes with the action *sync* in their alphabets must all perform this action before any of them can proceed. Implement a monitor called `Barrier` in Java with a method `sync` that ensures that all of *N* threads must call `sync` before any of them can proceed.

5.5 *The Savings Account Problem:* A savings account is shared by several people. Each person may deposit or withdraw funds from the account subject to the constraint that the balance of the account must never become negative. Develop a model for the problem and from the model derive a Java implementation of a monitor for the savings account.