Exercise 5: Synchronisation mechanisms in Java.  
Semaphores and monitors.

Hand-out: 2 May  
Due: 9 May

The goal of the exercise is to get acquainted with different synchronisation patterns commonly used in real-life applications and see how these patterns can be implemented using the concurrency primitives presented in the lecture. Several properties of concurrent computation, such as mutual exclusion, absence of deadlock and starvation, will be discussed and illustrated with simple examples.

1. **Producer – Consumer example**  
The producer - consumer synchronisation pattern is very common in many applications.

*Producer objects store elements in a shared bounded buffer; consumer objects consume elements from the same buffer.*

We will have a look at two different implementations of the producer-consumer scenario in Java: one based on the semaphores and the other based on monitors.

**Questions:**
1.1. Which implementation provides more guarantees (mutual exclusion, fairness, absence of deadlock)?
1.2. How would you re-implement the shared buffer (bbmo.java) so that it can be accessed by multiple producers and consumers?
1.3. What synchronisation mechanism(s) would help you solve the problem more easily?

2. **Dining Philosophers example**  
The dining philosophers problem is a popular scenario that illustrates the use of shared resources and the need for several guarantees: mutual exclusion, absence of deadlock, absence of starvation.

*n philosophers are sitting at the table with a large bowl of spaghetti in the middle and n forks distributed around the table in such way that there is one fork to the left and to the right of each philosopher. The spaghetti is so entangled that each philosopher needs two forks to eat it. A philosopher can only use the forks that are immediately to his left and right. A philosopher repeatedly executes the following sequence of actions: think; take forks from the table; eat; put forks back on the table. Usually, a
philosopher knows very well how to think and eat but the operations of acquiring the forks and relinquishing them have to be performed in such a way that no philosopher is starved to death (or dies from overweight).

We will have a look at different implementations of the dining philosophers example in Java. The examples illustrate different approaches to solving the problem of deadlock and starvation.

**Questions:**

2.1. What real-life synchronisation problems can be “mapped” to the dining philosophers problem?

2.2. Which implementation provides more guarantees (mutual exclusion, fairness, absence of deadlock) and how are they achieved?

2.3. What mechanism(s) would you like to have in the language in order to solve the problem more efficiently?

**Have fun!**