INX370, *Software Development*, Semester 1, 2014
Practical 10 — Understanding Threads

Solution to the Dining Philosophers Problem Exercise
The `Philosophers` class is the driver class for the problem and all three classes were originally given for the lecture. `Fork` has had an accessor added to determine if the Fork is available for pick up. The `run` method in `Philospher` has been altered to call one of `runOriginal()` [the code given in the lecture], `runRandom()` [implements random selection of the first fork to pick up] or `randomCheckFirst()` [checks to see if the second fork is available].

Solution to the Question of Priorities Exercise
The code is available in the Java file `RunThreeThreads` which is available in the zip. General observations from are given below *in italics*,

Run the programme several times and note the behaviour. Check the results when you vary MAX from say 20 to 10 and up to 50. Are consistent patterns observable?

Generally the thread started first gets the lion's share of the time slices early on, and so tends to complete earlier than the others. But this is not guaranteed, and some swaps may be observable. Nothing can be taken for granted.

What is happening? Relate your answer not merely to the actual priorities being assigned, but also to their relative magnitudes.

The priorities are equal, so the chances of a swap depend on the time that each thread actually takes to execute.

Now change the priority of `one` and `three` to `Thread.MIN_PRIORITY`. Note that this time there is one clear winner in the priority battles. Is this respected by the JVM?

No, not really. More consistency is observed for larger values of MAX. In practice we see the higher priority thread being 'largely' scheduled earlier, but this is not at all guaranteed. It is more likely for larger values of MAX that a majority of the relevant cases will be executed early.

Try additional variations, using a random number generator and/or the `sleep()` method to schedule changes to thread priorities. Try to explain your results.

According to the specification, a major difference in thread priority should result in pre-emptive scheduling: i.e. we dump the low priority thread and replace it with another of higher priority. Do we always see this behaviour? If not, what is happening and can we fix it?

Using explicit thread sleeps and joins is the only way of ensuring reproducible behaviour. The thread scheduler will almost never be free of OS-dependent influences.