INBN370, *Software Development*, Semester 1, 2009 Practical 8 (Week 9) — Understanding Threads

The classes for this week are about Threads, `Thread` objects, inheriting from the `Thread` class, and implementing the `Runnable` interface. Mostly, we are interested in you playing with the code from the lecture or perhaps something similar. In this prac session we focus on thread code that you have seen before, with one exception – we are going to spend some time playing with thread priority and see whether it works as advertised.

Exercise – Some Basic ThreadWork
In this exercise, we are going to take some of the code from the lectures and run it. Then we are going to get some very necessary, if slightly trivial, practice in implementing some of the examples using the interface `Runnable` instead of inheriting from `Thread` and vice versa.

Some tasks:

- Download the lecture code, compile and run all the demos shown in the class.
- Change `Packer.java` so that it implements Runnable rather than extends `Thread`. Make the necessary changes to `ThreadedFillDemo` so that the program creates a `Thread` from `Packer`.
- Choose another threaded example which uses `Runnable` and convert it to work by subclassing `Thread`
- Change some of the data and times in `ScheduleTasks` to see the effect of the `Timer` class.
- Alter the delay times on the messages in `JumbledMessage` to see the effect, in both the unsynchronized and synchronized versions.

Exercise – Dining Philosophers
Run the code provided in the lecture for the Dining Philosopher's problem, and see what responses appear. Try it three or four times so that you get a feel for the programme and some of the difficulties which emerge.

Now try to fix this source code. The fix will have to take place in **Philosopher.java**. There are different ways to fix this:

- Randomize the selection of the first fork. Use a random number generated by an instance of the Random class.
- Check if the second fork is available. If not put down the first fork. Does this lead to any other possible problems?

If you are interested in pursuing this problem further, keep track of the number of times that each philosopher gets to eat. Is the sharing of the food fair i.e. do they each get to eat about the same number of times?

Exercise – A question of Priorities
In this exercise we are going to examine the effect of setting priorities in threads. Begin by stealing from the example code to write a new class called `RunThreeThreads` which launches three very simple threads. We will call them `MsgThread`s, and all they will do will be to print out their name a defined number of times. We will also vary this number – it will be the same for each thread – to see how much of a difference it makes.

So in practice we need an additional class called `MsgThread`:

- Create this class by subclassing `Thread`.
- Ensure that `MsgThread` has a constructor which takes a string and calls the corresponding superclass constructor.
- Define an integer constant called *MAX* which will control the number of times each thread prints out its own name. Use a value of 20 to begin with.
- Implement a run method to do this through a simple for loop – the name should be printed to stdout.

Now return to the `RunThreeThreads` class. Make sure that you create three threads of type MsgThread, and that you give them obvious variable names like `one`, `two` and `three`, and that you use the constructor to set a similar display name. Then set the thread priority to its maximum value as shown:

```
Thread one = new MsgThread("One");
one.setPriority(Thread.MAX_PRIORITY);
```

Repeat this process for each of the threads created, and then successively call the three `start` methods:

```
one.start();
two.start();
three.start();
```

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?

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

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?

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?