CS4/MSc Distributed Systems
Tutorial 4: Java Threads

Stephen Gilmore
stg@dcs.ed.ac.uk

Tutorial: M 12, Room 5325
Office hour: Th 2, Room 1608

Yussuf Abu-Shaaban
yna@dcs.ed.ac.uk

Tutorial: M 12, Room 6207
Office hour: M 4, Room 3311

Threads: Java's concurrency package

- In many applications it can be natural to think of several *threads* of execution within a program, each making progress on its work.
- Using threads is an example of *concurrency* in a program.

$$\left.\begin{array}{ccc}\text{Thread 1} & \text{Thread 2} & \text{Thread 3}\\ \downarrow & \downarrow & \downarrow\end{array}\right\}\text{threads progress concurrently}$$

A sequential program has only a single thread of control.

- It may be useful to structure the program in this way even if ultimately the threads are *multi-tasked* onto a single processor.

Concurrency and parallelism

- The concurrent programming paradigm can be contrasted with the *parallel programming* paradigm where several physical processors are available.
- The use of parallelism in the program is to speed the execution of the program by making use of all of the available processors.
- In contrast, multi-tasking threads onto a single processor might actually slow the execution of the program.
- However, it provides a more natural programming environment for the application programmer and thereby reduces the potential for error.

Priorities and daemons

- Every thread has a priority, which can be set and inspected. Threads with higher priority are executed in preference to threads with lower priority.
- Each thread may or may not also be marked as a *daemon*. The Java Virtual Machine exits when the only threads running are all daemon threads.
- The `setDaemon()` method must be called before the thread is started.
- The `isDaemon()` method can be used to test whether or not a thread is a daemon thread.

Threads in the class hierarchy

```
java.lang.Object         (equals, getClass, toString, ...)
      ↓
java.lang.Thread         (run, sleep, yield, interrupt, destroy,
                          isAlive, getPriority, setPriority, ...)
```


Creating threads (method 1)

- There are two ways to create a new thread of execution. One is to declare a class to be a subclass of `java.lang.Thread`.
- This subclass should *override* (re-define) the `run()` method of class `Thread`.
- An instance of the subclass can then be allocated with `new` and started using the `start()` method.
- Note that the `run()` is special, like `main()`. We define `run`, but we invoke `start` (which causes `run` to be invoked later).

Creating threads (method 2)

- The other way to create a thread is to declare a class that implements the `java.lang.Runnable` interface.
- That class then implements the `run()` method.
- An instance of the class can then be allocated, passed as an argument to one of the constructors of class `Thread`, and started using the `start()` method.
- In both cases the `run()` method is a public void method

```
public void run()
```


A simple thread

```
class Odds extends Thread {
    private int odd = 1;
    public void run() {
        while (true) {
            System.out.println(odd);
            odd += 2;
        }
    }
}
```

Using threads

```
class ThreadExample {
    public static void main (String[] args) {
        Odds o = new Odds();
        Evens e = new Evens();
        o.start();
        e.start();
    }
}
```


Thread synchronisation

- In object-oriented programming the local state of an object is typically stored in private fields, modified in a controlled way through accessor methods.
- If it would be problematic for more than one thread at a time to invoke a method the method can be marked with the access modifier `synchronized`.
- A client arriving to inspect a value before it is ready would be made to wait (with `wait()`) and then later would be notified (with `notify()`).

Locks and monitors

- Each object contains a single *lock* (also called a *monitor*) which is acquired by the first thread to invoke a synchronised method on that object. No other synchronised method can be invoked until the first thread finishes and releases the lock. Static synchronised methods lock the `Class` object for a class.
- Making a method invocation synchronised imposes a run-time overhead so some classes are *thread-safe* and some are not. Many of the collection classes in the Java Collection Framework are not thread-safe.

Deprecated thread methods

- The thread methods `stop()`, `suspend()` and `resume()` have been deprecated since Java 1.2. The use of these methods was found to encourage poor concurrent programming style.
- Use of the `stop()` method on a thread causes it to unlock all the objects that it has locked. If any of these were in an inconsistent state, other threads may now view these objects in an inconsistent (or *damaged*) state.
- Use of the `suspend()` method is very likely to cause a *deadlock*. If the target thread holds a resource when it is suspended, no thread can access the resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock the resource prior to calling resume, deadlock results.