ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

THREADS AND CONCURRENCY

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ucl.ac.uk

Concurrency

- **Concurrency** is a property of running programs to execute computations “in parallel”, overlapped in time
  - These computations share common resources as they have access to the same memory space
- Concurrent use of shared resources is the source of many difficulties
  - Common resources may be accidentally corrupted or get into an inconsistent state when accessed concurrently
- The design of programs to deal with concurrency entails finding reliable techniques for coordinating the execution, data exchange and common memory access among concurrent execution threads

Single and Multi-Threaded Programs

- Most programs have a single thread of execution, with commands executed sequentially starting from the main procedure or method
  - If the program performs many activities, a relatively long activity may delay other activities, e.g. reading new keyboard input, thus making the program unresponsive
  - Also some method calls block until something is available to process, e.g. waiting for a network connection, which means nothing else can be done in the meantime
- With concurrency, the basic thread of execution starts from the main method but additional threads can be spawned at any time during the program execution, as and when required
  - All the running threads have access to the same memory space, so care is required regarding access to common data
  - If a thread terminates abnormally, e.g. because of an exception, the rest continue and may depend on it -> important to catch exceptions!

(Pseudo-) Parallel Execution

- Concurrent tasks realised as threads execute in a "pseudo-parallel" fashion, with the operating system scheduler giving them distinct "CPU time-slices"
  - This is in exactly the same fashion in which the scheduler executes multiple running programs on single-processor systems
- True parallel execution requires threads that exhibit inherent parallelism, i.e. what each thread does is well independent of each other, and a multi-processor system
  - In this case, each processor runs a different thread and threads are swapped in an out in and out of the processors
- In today's multi-processor systems, typically whole programs rather than threads are delegated to different processors

Concurrent Application Examples

- When downloading and executing audio and video clips over the Internet, the application does not need to wait for the full download before starting playback
  - In this case, synchronisation is required between the two threads
- In a word processor, while a large file is (periodically) saved to disk, the user can continue to word-process “in parallel”
- While a Java program runs, the runtime system may start a thread to reclaim unused memory or otherwise perform “garbage collection” while the program is running
- An application listening for new network connections through a blocking read in the main thread, spawns a new thread to deal a every new connection

Programming Languages and Concurrency

- Initially, support for concurrent programming was only available in an operating system's kernel environment
- **Ada** was the first programming language to support concurrency but it is not a widely-used language
  - It was developed by the US Department of Defence and has been mostly used for military command-and-control systems
- **C** and **C++** use platform-specific libraries for concurrent programming e.g. POSIX threads
  - Porting may be required between different operating systems
- **Java** integrates concurrency support in the core language through threads of execution, giving powerful capabilities to the programmer that are available across systems
  - Highly portable in the Java philosophy

Creating and Executing Threads

- The Java thread API has changed in J2SE 5.0, although the old and simpler API is still available
- In J2SE versions < 5.0, threads are supported through the **Thread** class and its methods **start** and **run**
  - **This is the method we will be using**
- In J2SE 5.0, threads are supported through the classes **Executors** and its method **newFixedThreadPool**, **ExecutorService** and its method **execute** and finally the interface **Runnable** and its method **run**
  - This is a more powerful but also more complicated method

The Thread Class

- The Thread class exhibits the key methods start and run
- Any activity that needs to run as a parallel thread has to be implemented as the method run of a specific thread class that extends (i.e. inherits from) class Thread
  - E.g. method run of class PrintThread in the example to follow
- A thread (in addition to the default main program thread) can be started by creating a new thread object and calling its start method
  - E.g. we are starting three objects of class PrintThread in the example to follow
- The Thread class also supports a sleep method that will make the thread inactive ("sleeping") for a number of milliseconds
  - Note: a subclass inherits all the methods of the superclass, e.g. PrintThread inherits all the methods of Thread such as start, run, sleep and getName

A Simple Multi-Threaded Program - 1

```
import java.util.Random;

class PrintThread extends Thread {
    private int sleepTime;

    public PrintThread (String name) {
        // first we call the superclass constructor
        // Thread class keeps a private String name instance variable
        super(name);

        // we now initialise the sleepTime to a random time up to 5 secs
        // this private instance variable will be used in the run method
        Random randGen = new Random();
        sleepTime = randGen.nextInt(5001); // 0 .. 5000 msecs
    }

// continues on the next page
```

A Simple Multi-Threaded Program - 2

```
// continues from the previous page

    @override // override Thread's run method (does nothing in Thread)
    public void run () {
        try {
            System.out.printf("%s going to sleep for %d msecs\n",
                                    getName(), sleepTime);
            Thread.sleep(sleepTime);
            System.out.printf("%s done sleeping, ends\n", getName());
        }
        catch (InterruptedException e) {
            System.err.printf("%s interrupted while sleeping\n", getName());
            e.printStackTrace();
            // printStackTrace is a method of class Throwable from which
            // all exception classes inherit
        }
    }

} // end class PrintThread

// continues on the next page
```

A Simple Multi-Threaded Program - 3

```
// continues from the previous page

public class ThreadTester {

    public static void main (String args[])
    {
        PrintThread thread1 = new PrintThread("thread1");
        PrintThread thread2 = new PrintThread("thread2");
        PrintThread thread3 = new PrintThread("thread3");


        System.out.printf("starting threads 1, 2 & 3\n");
        thread1.start(); // will eventually invoke run
        thread2.start(); //  ..
        thread3.start(); //  ..
        System.out.printf("threads scheduled to start, main ends\n");
    }

} // end class ThreadTester and end of program
```

Some Observations

- The previous example demonstrates threads 1, 2 & 3 started from the main program which sleep for a random time, hence threads 2 and/or 3 may finish before thread 1
  - In fact the program has 4 threads, including the main thread
- It should be noted that the main thread may finish before any of the other three threads start executing:
  - The Thread start method puts a thread in a “ready to execute state” but the actual execution start depends on the scheduler
  - In fact, threads can also be given priorities between 1 (min) and 10 (max), with the default value being 5 (norm); threads with higher priority execute first
- You are advised to run this program in order to see the threads and their execution and ending in action

Synchronisation Requirements

- Multiple threads can have access to common objects, e.g. those created in the main thread and passed to a new thread as constructor arguments
  - This can lead to discrepancies and inconsistencies and subsequently compromise program integrity
- A typical case is based on “read-write” interactions
  - Let’s assume that an instance variable is read, a value is added to it and the instance variable is updated – read and write are implemented through getValue and setValue methods
  - E.g. Thread 1 reads common object value which is 1000 and adds 200 while Thread 2 does the same and adds 10
  - If both threads do this almost at the same time, Thread 1 will make the value 1200 and then Thread 2 will make it 1010, instead of the correct value that should be 1210 after the two transactions
- The commonly accessed instance variable ends up with a corrupted, i.e. incorrect, value

Synchronisation with Lock

- Java uses locks to perform synchronisation. Any object can contain an object that implements the Lock interface, typically the ReentrantLock one.
- A thread that accesses the containing object needs to do the following to achieve synchronised access:
  - It should first call the Lock's **lock** method; if another thread has the lock, this one will be put in a waiting state
  - It should then call the getValue and setValue methods; the thread will be allowed to proceed by the runtime system only if the object is not locked by another thread
  - It should finally release the lock by calling the Lock's **unlock** method; forgetting to do this will block forever any other waiting threads!

Example Synchronisation with Lock

```
import java.util.concurrent.locks.*;

class SynchronisedReadWrite {
    ReentrantLock accessLock = new ReentrantLock();
    private int value;

    public int getValue () {
        return value;
    }

    public void setValue (int newValue) {
        value = newValue;
    }
}

    // …  - srw is a SynchronisedReadWrite object and x the value to add
    srw.accessLock.lock();
    int srwValue = srw.getValue();
    srw.setValue(srwValue + x);
    srw.accessLock.unlock();
    // …
```

Summary

- Java threads provide powerful support for concurrency in a platform-independent fashion
- The Thread class provides support for concurrency with its start and run methods
  - Its also supports getName and sleep methods
- When using threads, locking may be required to lock “critical program regions” that require exclusive access by a single thread