# Programming in Java

## Threads and multi-threading — the Fork/Join API


KLM

Department of Computer Science and Information Systems
Birkbeck, University of London

`keith@dcs.bbk.ac.uk`


# High Level Concurrency Objects

As concurrency was seen as too low level in previous versions of Java some new classes were introduced in

**Lock objects** support locking idioms that simplify many concurrent applications

**Executors** define a high-level API for launching and managing threads

**Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronisation = good!

**Atomic variables** have features that minimise synchronisation and help avoid memory consistency errors

`ThreadLocalRandom` provides efficient generation of pseudo-random numbers from multiple threads

# Executors — `ForkJoin` I

- The use of `ForkJoinPool` and `RecursiveTask`
- Using the fork/join framework is straight forward
- The first step is to write some code that performs a segment of the work
- Your code should look similar to this:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

# Executors — `ForkJoin` II

- wrap this code as a `ForkJoinTask` subclass
- typically as one of its more specialised types
  - `RecursiveTask` (which can return a result) or
  - `RecursiveAction`
- After your `ForkJoinTask` is ready
- create one that represents all the work to be done and pass it to the `invoke()` method of a `ForkJoinPool` instance

# Executors — ForkJoin III

```
package helloWorld;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Incrementor extends RecursiveTask<Integer>{
  ...
}
```

- Next you need to create a class that extends `RecursiveTask`
- A `RecursiveTask` is like a `Thread`
- But you can retrieve a value from it after it finishes
- Supply a type parameter (`Integer`, in this example) to specify the kind of value you want

# Executors — ForkJoin IV

```
package helloWorld;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class Incrementor extends RecursiveTask<Integer>{
  public static ForkJoinPool fjPool = new ForkJoinPool();
  ...
}
```

- Then, create a `ForkJoinPool`
- Create only one of these, and it should be `static`
- A `ForkJoinPool` is a pool of `Thread` objects.
- Java will take care of all the thread allocation and deallocation for you

# What did that code do? I

- Is that any different from a standard method call?
- Answer: yes, `compute`, which replaces `run`, runs as a `RecursiveTask`
- So what?

# What did that code do? II

- `RecursiveTask`s are, so-called, `lightweight` threads (you can have 32K of them) which can happily `fork` and `join`
- `public final ForkJoinTask<V>fork()` arranges to asynchronously execute this task
  Note: any given task should only be forked once
- `public final V join()`, unlike the `Thread join`, this returns a result

# Another example I

Look at the code. . .

# Another example II

- The call to `fjPool.invoke` creates an instance of this `Sum` class, which is a `RecursiveTask`, which gets things started
- `invoke` should not be called from within a `RecursiveTask` or `RecursiveAction`
- It should only be called from sequential code

# Atomic variables

- The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables
- All classes have `get` and `set` methods that work like reads and writes on volatile variables
- a `set` has a `happens-before` relationship with any subsequent `get` on the same variable
- The atomic `compareAndSet` method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables

Example...

# We've barely scratched the surface. . .

So please do follow up the links on the Moodle site

# Questions