Objectives

- Task with Concurrency Utilities
  - ThreadPools
  - Latches
  - Future and Callable

Usage of Executor Service


```
ExecutorService service = Executors.newFixedThreadPool(10);
Future future = service.submit(new MyCallable());

future.get();
```

- `ExecutorService` is an interface for dealing with concurrency from `java.util.concurrent`
- There are many implementations of the executor service interface that use thread pools (here we use a fixed size thread pool)
- The `future` is returned asynchronously and represents a pending computation
- `.get()` blocks until the result is available

CountDownLatch


```
// Creation of latch
// Counter initialized to N
CountDownLatch latch = new CountDownLatch(N);

    // Within task
    // After completion call...
    latch.countDown();

// Await completion of all tasks
latch.await();
```

- A `CountDownLatch` can be used to synchronize tasks executing in different threads
  - For example, block until all tasks are finished
- A latch is a basically counter that we decrement every time one of our tasks completes

Task

- Create a concurrent program that calculates primality of integers from $(1, 10000000)$
- Create a task `Callable<Boolean[]>` that looks at a range of natural numbers, from `start` to `end` and returns an array of boolean values to indicate which numbers in the range are prime
  - Use the `isPrime()` method from `org.apache.commons.math3.primes` to test for primality
- Do this task in parallel by dividing the range $(1, 10000000)$ into intervals of length 2500000
- Coordinate the completion of the threads using a `CountDownLatch`
- Compare performance to the single-threaded case
- Write your solution in a single java source file and submit to the moodle