Objectives

- Talk about concurrency basics
- Java threads and monitor Locks
- `java.util.concurrent`

Threads [1]

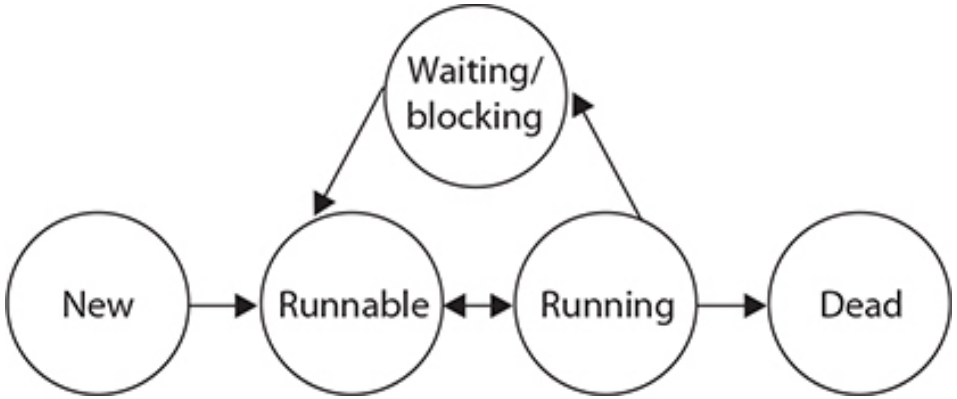

Figure: Thread state machine diagram, from [1]

- *Thread* has two meanings in Java
  - An instance of `java.lang.Thread`
  - A thread of execution, lightweight process that has a call stack
- A thread can be in one of *five* states
  - New
  - Runnable
  - Running
  - Waiting, blocked, sleeping
  - Dead

Threads [1]

- The JVM has its own mechanism for scheduling its processes: *Thread Scheduler*
  - Not necessarily a one-to-one correspondence to OS processes
- A thread is defined in one of two ways
  - Extend `java.lang.Thread`
  - Implement the `Runnable` interface
- In either case, the main task is to implement the `run()` method
  - Note that overloading `run` probably will not result in the behavior you expect
- With multi-threading it is important that there are few guarantees, order of execution for example

Thread Methods [1]

- The `Thread` class has methods for creating, starting, and pausing threads
  - `start()`: When the thread is instantiated it is in the *new* state, calling this method on a `Thread` instance moves it from *new* to *runnable*
  - `sleep()`: `static` method that causes the current thread to pause a given number of milliseconds
  - `yield()`: supposed to allow threads to take turns based on *priority*, but behavior is not guaranteed
  - `run()`: instance method that defines the work that will be done in the thread
- There are also overloaded versions of the `Thread` constructor
  - `Thread()`
  - `Thread(Runnable r)`
  - `Thread(Runnable r, String name)`
  - `Thread(String name)`

Deprecated Thread Methods [1]

- If you open your IDE and point at a reference to a `Thread` object you will notice that there are a number of deprecated methods
- These methods were used prior to Java 2
  - `suspend()`
  - `resume()`
  - `stop()`
- Can use a flag variable on the thread and the `wait()` and `notify()` methods (described below) to suspend a thread

Joining Threads [1]


```
Thread myThread = new Thread();
myThread.start();
myThread.join();
```

- `join()` is an instance method of the `Thread` class
- Used in a situation when we know that one thread needs another thread to be finished
- Suppose the example above is run from the main thread, then the main thread is joined to `myThread`, the main thread will be blocked until `myThread` finishes
- Also an overloaded version that allows the programmer to specify a timeout

Locks and `synchronized`

- The `synchronized` keyword allows the Java programmer to enforce atomicity
- A `synchronized` block consists of two parts
  - An object that serves as a lock
  - A block of code that is guarded by the lock
- A `synchronized` method simply means a block that spans the entire body of the method
- Every object in Java has a built-in, called the *monitor lock* that controls access to its synchronized methods
- Static methods can also be synchronized, and there is one lock for the entire class (instance of `java.lang.Class`)

Safety and Liveness

- We can encounter problems with multi-threaded applications when there is shared mutable state, multiples threads accessing and updating the same data
- `synchronize` our code to prevent common pitfalls
- Desirable properties of concurrent systems
  - *Liveness*: system is always able to make progress
  - *Safety*: bad things don't happen
- *Deadlock*: Two threads are blocked, each waiting for the other's lock
- *Starvation*: Thread unable to make progress because a resource it needs is continually being used by other threads

Interprocess Communication [2]

- Java provides methods (available on each object) that can be used only in a synchronized context and allow for inter-thread communication
  - `wait()`: indicates that calling thread should give up the lock and go to sleep
  - `notify()`: wakes up a thread that called `wait()` on the same object
  - `notifyAll()`: wakes up all threads that called `wait()`
- Deprecated methods

Producer-Consumer


```
public synchronized String take() {
    // Wait until message is
    // available.
    while (empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = true;
    // Notify producer that
    // status has changed.
    notifyAll();
    return message;
}
```

- Producer-Consumer Example from the Oracle Documentation

Producer-Consumer


```
public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
```

- Producer-Consumer Example from the Oracle Documentation

Java Concurrency [2]

- We have seen that Java has good built-in support for synchronization and multithreading
  - Creating new threads
  - Interprocess communication with `wait()` and `notify()`
- These techniques are not necessarily suitable for programs that make *extensive* use of multithreading
- With JDK 5 introduced the concurrency utilities, also called the concurrent API
- Offered a number of high-level features for concurrent application programming
- JDK 7 added even more utilities including the Fork/Join Framework
- The concurrency utilities are defined in `java.util.concurrent`

java.util.concurrent

- Concurrent features are organzied as follows
- *Synchronizers*
- *Executors*
- *Concurrent collections*
- *Fork/Join Framework*

Synchronizers [2]

- High-level way of synchronizing interactions between multiple threads
- *Semaphore*: classic semaphore
- *CountDownLatch*: Wait until a specified number of events have happened
- *CyclicBarrier*: Enables a group of threads to wait at an execution point
- *Exchanger*: Exchange data between two threads
- *Phaser*: Synchronize threads that go through multiple phases of operation

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, wait for a number of events to occur before proceeding
  - For example, block until all tasks are finished
- A latch is a basically counter that we decrement every time one of our tasks completes

CyclicBarrier [2]

- A CyclicBarrier is helpful when two or more threads need to wait at a predetermined execution point
- We can instantiate a `CyclicBarrier(int numThreads)` and make sure that each of the threads holds a reference to the barrier
- Within each thread, we call `await()` on the barrier to indicate that that particular thread has reached the predetermined execution point

Executors [2]

- An *executors* initiate and control the execution of threads
- `ExecutorService` is an extension of the `Executor` interface (this is the one we mostly use)
  - Can handle threads that return results

Tasks

- Concurrent programs handle many tasks
  - *Task*: discrete unit of work
- To organize a concurrent program one should first identify *task boundaries*[1]
  - In the best case, tasks are independent (see also famous quote from Brooks[2])
- A given kind of application will often suggest natural task boundaries
  - For example, in a server application, it may be natural to divide tasks per request
  - The Executor framework in Java gives us many tools to handle concurrent tasks

---

[1] B. Goetz, T. Peierls, D. Lea, *et al.*, *Java concurrency in practice*. Pearson Education, 2006.

[2] F. P. Brooks Jr, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.

Java Executor Framework

- The Executor framework consists of a number of interfaces and implementations: `Executor`, `ExecutorService`, `Callable`, `Future`, ...
- *Threadpool*: manages and maintains multiple thread objects, amortizes the cost of thread creation over the entire running time of your application
  - Real-life application: call-center
- Threadpool size can vary depending on the type of application, e.g. I/O versus computation, and the number of cores
  - Fixed-size or Cached

Java Executor Framework

- `Runnable`, `Callable`, and `Future` help us manage tasks and asynchronous computation
- `Runnable`: Task that does *not* return a result, "one-way" task
- `Callable<T>`: Task that can return a result, "two-way"
- `Future<T>`: Like a JS promise, represents the result of an asynchronous computation
  - Indicate the return type with a generic parameter
  - Also, may throw exception

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Threadpool Executor


```
public static void concurrencyExamples( ) throws
                                        InterruptedException,
                                        ExecutionException {
  ExecutorService service = Executors.newSingleThreadExecutor();

  Future<String> future = service.submit(new Callable<String>() {
        @Override
        public String call() throws Exception {
            return "Callable hello world";
        }
  });

  // Alternative with lambda expression
  future = service.submit(() -> "Hello, world!");

  System.out.println(future.get());
}
```

- Code shows use of a particular ExecutorService obtained from a static factory method
- Also shows that we can pass in a lambda expression for a Callable

Java Executor Framework

- Definition of `newSingleThreadExecutor` from Java documentation

```
Creates an Executor that uses a single worker thread operating off an
unbounded queue.  (Note however that if this single thread terminates
due to a failure during execution prior to shutdown, a new one will take
its place if needed to execute subsequent tasks.)  Tasks are guaranteed
to execute sequentially, and no more than one task will be active at any
given time.  Unlike the otherwise equivalent newFixedThreadPool(1) the
returned executor is guaranteed not to be reconfigurable to use
additional threads.
```

Reminder

- We will share a schedule for the sprint review
- Please make sure that someone enters your team in one of the time-slots

References I

[1] K. Sierra, B. Bates, and E. Robson, *OCP Java SE 8 Programmer II Exam Guide (Exam 1Z0-809)*. McGraw-Hill Education, 2018.

[2] H. Schildt, *Java: The Complete Reference, Eleventh Edition*. McGraw-Hill Education, 2018.

[3] B. Goetz, T. Peierls, D. Lea, J. Bloch, J. Bowbeer, and D. Holmes, *Java concurrency in practice*. Pearson Education, 2006.

[4] F. P. Brooks Jr, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India, 1995.