Concurrency in Java
What is a thread anyway?

- Smallest sequence of instructions that can be managed independently by a scheduler
- There can be multiple threads within a process
- Threads can execute concurrently and share resources

1. "process" in the operating system sense
What is a thread anyway?

- Smallest sequence of instructions that can be managed independently by a scheduler
- There can be multiple threads within a process
- They can execute concurrently and share resources

Discussion

In Java, who is the scheduler?
Who is the process?
What are the resources?
### Threads in Java

In Java, threads are objects with (among other things)

<table>
<thead>
<tr>
<th>priority</th>
<th>higher priority executed in preference of lower priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>unique and does not change</td>
</tr>
<tr>
<td>state</td>
<td>Thread.State enumeration: {NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED}</td>
</tr>
<tr>
<td>lock</td>
<td>unique</td>
</tr>
<tr>
<td>stack</td>
<td>unique</td>
</tr>
<tr>
<td>run()</td>
<td>a method that takes no args and returns <strong>void</strong></td>
</tr>
</tbody>
</table>
Creating threads

There are two ways and we will see both:

1. Extending the `Thread` class
2. Implementing the `Runnable` interface
1. Extending the **Thread** class

- **Thread** is a class in **java.lang**
- It has a **run()** method
- A class can be made into a thread by
  - extending **Thread** and
  - implementing **run()**
1. Extending the **Thread** class

Example

class **PrimeThread** extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger than minPrime
    }
}

Create and run the thread:

PrimeThread p = new PrimeThread(143);  
p.start();
2. Implementing the `Runnable` interface

- `Runnable` is an interface in `java.lang`.
- It has a single method `run()`.
- A class can be made into a thread by
  - implementing `Runnable`, i.e. implementing `run()`, and
  - passing an instance of the class to a `Thread` constructor.
2. Implementing the **Runnable** interface

Example

```java
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
    }
}
```

Create and run the thread:

```java
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```
Advantage of implementing `Runnable` over extending `Thread`:

- Since Java has single inheritance, a class can't inherit from `Thread` and from another class at the same time.

- Implementing `Runnable` allows you to
  - get a class with thread-line behavior and
  - inherit from a super class at the same time.
Abstract classes Vs Interfaces

In Java, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

If Java allowed for multiple inheritance, the distinction between Abstract class and Interface would be more blurred.
Some side tracking...

Interface

- A group of related methods with empty bodies.
- A class can implement an interface
  - give source code to all interface methods

Discussion

- Both class and interface form a contract with the outside world. How are these *contracts* different?
More side tracking...

Abstract class

- A class with empty methods that are declared abstract.
- An abstract class cannot be instantiated.
- An abstract class can be extended (i.e. subclassed)
  - Subclass remains abstract until all abstract methods are overridden, i.e., given a concrete implementation.

Discussion

How is an abstract class similar to an interface?
How are they different?
More side tracking...

- They are implemented in `java.lang` package
  - `java.lang` exists since JDK 1.0 and is imported by default
  - still, threads are not part of the language proper

Discussion

Threads in `java.lang` as opposed to in the "language proper" was a design decision by James Gosling et al. What are the advantages and disadvantage of this decision?
The **Thread** class

```java
public class Thread extends Object implements Runnable
```

- `static currentThread()` returns current running thread
- `getId()`
- `getState()` can be one of `{NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED}`
- `interrupt()`
- `join()` waits for this thread to die
- `sleep()`
- `start()`
- `yield()` useful for implementing co-routines
Synchronization

Two basic synchronization idioms in Java:

1. Synchronized method
2. Synchronized statement/block
1. Synchronized method

Used to enforce two things:

1. Mutual exclusion
   - Prevent interleaving of two invocations of synchronized methods on the same object.

2. Ordering & visibility
   - "When a synchronized method exits, it establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object."
Synchronized statement/block

- We can synchronize only parts of a method as opposed to the whole method.

```java
public void add(int value){
    ...
    synchronized(this){
        this.count += value;
    }
    ...
}
```
java.util.concurrent

- Package with threading utilities, for example:
  - Concurrent collections
  - Atomic variables
  - Synchronizers
  - Locks
  - Timing
- Frees the programmer from creating these utilities by hand
Object as Monitor

The `Object` class has

- a unique lock, which we can use as a `condition variable`

- a `wait()` method

- `notify()` and `notifyAll()` methods

That looks a lot like a monitor...
Object as Monitor

The `Object` class has

- a unique lock, which we can use as a *condition variable*
  - limited to one condition variable
- a `wait()` method
- `notify()` and `notifyAll()` methods

That looks a lot like a monitor...
Object as Monitor

The `Object` class has

- a unique lock, which we can use as a condition variable
  - limited to one condition variable
- a `wait()` method
  - ordering of `wait queue` is implementation dependent
- `notify()` and `notifyAll()` methods

That looks a lot like a monitor...
Object as Monitor

The `Object` class has

- a unique lock, which we can use as a *condition variable*
  - limited to one condition variable
- a `wait()` method
  - ordering of *wait queue* is implementation dependent
- `notify()` and `notifyAll()` methods
  - *signaling discipline* is fixed at Signal & Continue

That looks a lot like a monitor...
Object as Monitor

The `Object` class has

- a unique lock, which we can use as a *condition variable*
  - limited to one condition variable
- a `wait()` method
  - ordering of *wait queue* is implementation dependent
- `notify()` and `notifyAll()` methods
  - signaling discipline is fixed at Signal & Continue

That looks a lot like a monitor... *a very simple one.*
Semaphore implementation in Java

```java
public class Semaphore {
    private int value;

    public Semaphore(int initial) {
        value = initial;
    }

    synchronized public void up() {
        ++value;
        notifyAll();
    }

    synchronized public void down() throws InterruptedException {
        while (value == 0) wait();
        --value();
    }
}
```
Other interesting stuff...

Weak memory models

- What's a memory model anyway?
- We will touch on that in a future class, but not relate it back to Java too much.
- Java's memory model is complex and a hot topic of debate.
Other interesting stuff...

Albeit not covered by this class

- `volatile` keyword
- `Lock` class
- thread groups
- thread as daemon
- Remote Method Invocation (RMI)
  - an object in a Java Virtual Machine invoking methods on object in another JVM
  - falls into a *Distribution vs Concurrency* discussion
Last but not least,

Please also look at `main7java.pdf` from the course website.

There you'll find more examples.
[Java thread]
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
[Java concurrent]
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html