Java Threads & Concurrency
Online Reference

- [http://docs.oracle.com/javase/tutorial/essential/concurrency/](http://docs.oracle.com/javase/tutorial/essential/concurrency/)
Threads

• Threads are Objects too!

• http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

• You can also use Executors, but more on those later.
Thread Objects

```java
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

- Thread has an empty `run()` method, you can override it
- Starting the thread will run the `run()` method and immediately return -- it does not block like other methods
The Runnable Interface

- Similar to a `Thread` object, except an interface
- Use it to create new threads that need to extend another class

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```
Pausing a Thread

public class SleepMessages {
    public static void main(String args[]) throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };

        for (int i = 0; i < importantInfo.length; i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}

• The `sleep` method pauses a thread for (roughly -- OS dependent) that many milliseconds

• If another thread interrupts a sleeping thread, the sleep method will throw an `InterruptedException`
Interrupting a Thread

for (int i = 0; i < importantInfo.length; i++) {
    //Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        //We've been interrupted: no more messages.
        return;
    }
    //Print a message
    System.out.println(importantInfo[i]);
}

• Will print a message every four seconds until interrupted or there are no more messages
Interrupting a Thread 2

```java
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

- What if your methods don’t throw InterruptedException?

- `Thread.interrupted()` returns true if the current thread has been interrupted. A subsequent call to `Thread.interrupted()` will return false unless the thread was interrupted again.

- It may be better to throw a new InterruptedException instead of returning.
“Joining” a Thread

• `t.join();` will wait for the thread `t` to complete

• `t.join(millis);` will wait at most `millis` ms (again roughly) for `t` to complete

• if interrupted, will throw an `InterruptedException`
Synchronization

• Threads communicate by sharing access to fields and methods of objects they reference

• This can lead to some big problems
Thread Interference

class Counter {
    private int c = 0;

    public void increment() { c++; }
    public void decrement() { c--; }

    public int value() {
        return c;
    }
}

declare the c++ statement:
    retrieve c
    increment c
    store value

declare the c-- statement:
    retrieve c
    decrement c
    store value
What if two threads use the same Counter?

Thread A calls increment, Thread B calls decrement

1. Thread A: retrieve c (A’s c == 0)
2. Thread B: retrieve c (B’s c == 0)
3. Thread A: increment c (A’s c = 1)
4. Thread B: decrement c (B’s c = -1)
5. Thread A: store c (stores 1)
6. Thread B: store c (stores -1)
Thread Interference 3

• What went wrong?

• Performing operations on the same memory with multiple threads at the same time can cause some very nasty bugs
Memory Consistency

Thread A and B share a reference to counter:

    int counter = 0;

Thread A increments counter:

    counter++;

After, B prints out counter:

    System.out.println(counter);

B may print out 0!
Due to Threading implementations and hardware, A and B may not necessarily be working on the same memory.
Happens-Before relationships guarantee some statements happen before others

Thread.join() and Thread.start() are two examples

More Reading:

http://java.sun.com/javase/7/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility
Synchronized Methods

public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() { c++; }
    public synchronized void decrement() { c--; }

    public synchronized int value() {
        return c;
    }
}

- It is not possible for threads to interleave/interfere on a synchronized method -- only one thread may be executing the code synchronized on an object at a time, others will wait

- Synchronized methods establish *happens-before* relationships on subsequent method invocations

- Having a synchronized method is like wrapping a mutex around the method.

- Constructors cannot be synchronized -- so be careful
Synchronized Blocks

```java
public void addName(String name) {
    synchronized( name ) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- Will only synchronize the block on this
Synchronized Blocks 2

- Allows fine grained synchronization

- Be careful: if c1 and c2 were objects that shared references to other objects, they could interleave in other methods

```java
public class MsLunch {
    private long c1 = 0, c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) { c1++; }
    }
    public void inc2() {
        synchronized(lock2) { c2++; }
    }
}
```
Reentrant Synchronization

- Using synchronized gives threads a lock on a section of code.
- A thread cannot execute code another thread has a lock on.
- A thread can get a lock on code it already has a lock on, this is *reentrant synchronization*.
- Without this, it would be much easier to create deadlock (ex., a synchronized method calls itself).
Atomic Accesses

• An atomic action happens all at once (therefore they can’t interleave)

• Reads and write for reference variables and primitive (except long and double) are atomic

• Variables declared volatile also have atomic read and write (even long or double)

• Writing to a volatile variable also sets up happens-before dependencies to subsequent reads of that variable
Liveness

• A concurrent applications ability to execute in a timely manner (or at all) is its ‘liveness’

• Deadlock, starvation and livelock are concurrent programming issues which prevent liveness
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) { this.name = name; }
        public String getName() { return this.name; }

        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s has bowed to me!%n", this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s has bowed back to me!%n", this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
Deadlock 2

- Deadlock can happen if they both enter bow, then attempt to invoke bowback on each other.
- Both have a lock (from bow), and can’t obtain each others lock
Starvation

• Thread A obtains a lock another Thread B needs
• Thread A never releases lock, or computes for very long amounts of time with lock
• Thread B cannot progress (or progresses very slowly)
Livelock

- Threads can act in response to each other (via InterruptedExceptions, for example)

- Think of two people trying to pass each other in a hall.

- Thread A moves left and Thread B moves right (they still block each other)

- Thread A moves right and Thread B moves left (they still block each other)

- Both are still active, but neither can progress
Guarded Blocks

public void guardedJoy() {
    //Simple loop guard. Wastes processor time. Don't do this!
    while(!joy) {}  
    System.out.println("Joy has been achieved!");
}

• What if you want to wait for a field to be changed?
public synchronized guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}

• Be careful -- the InterruptedException might not be
  the one you were looking for -- put wait in a loop
• Why is guardedJoy synchronized? Can only call wait() when there is a lock (wait() releases the lock and suspends the thread)
Guarded Blocks 3

public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}

• **notifyAll()** notifies all threads waiting on a lock (and the scheduler decides which will get it next)

• **notify()** only notifies a single thread
Immutable Objects

- Immutable objects are objects that cannot change their state
- Very useful in concurrent programming -- since they cannot change state they cannot suffer from thread interference or have an inconsistent state
Immutable Objects 2

• It’s easy to make an object immutable in Java
• make all fields final (they can’t be modified) and private (they can’t be accessed)
• instantiate all fields within the constructor
• provide get methods for users to access copies of fields (primitives are naturally call-by-value)