Java Threading

- Threads are Objects, too
- Two approaches
  - java.lang.Thread
    - https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html
  - java.lang.Runnable
    - https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Thread Example

```
public class HelloThread extends Thread {

        private int val;

        public HelloThread(int val) {
                this.val = val;
        }

        public void run() {
                System.out.println("Hello from thread " + this.getName() +
                                ". My val is " + val);
        }

        public static void main(String[] args) {
                for (int i = 0; i < 5; i++) {
                        HelloThread hello = new HelloThread(i);
                        hello.start();
                }
        }
}
```

Runnable Example

```
public class HelloRunnable implements Runnable {

    private int val;

    public HelloRunnable(int val) {
        this.val = val;
    }

    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("Hello from thread " + name +
                ". My val is " + val);
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            Runnable runnable = new HelloRunnable(i);
            Thread thread = new Thread(runnable);
            thread.start();
        }
    }
}
```

Thread vs Runnable

- Extend Thread class
    - Simpler.
- Implement Runnable
    - More flexible.
- General rule: Program to Interfaces

Multi-threaded Server

```
while (true) {
    accept a connection
    create thread for client
}
```

Advantages

- Less work for main thread
- Multiple requests handled simultaneously
- Better responsiveness than single-threaded server

Caution

- Make sure tasks are thread-safe
- Does not limit the number of threads created

Multi-threaded Server

```
public class MultiEchoServer implements Runnable {
    Socket clientSocket;

    public MultiEchoServer(Socket client) {
        clientSocket = client;
    }

    public void run() {
        try {
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println(inputLine);
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try {
            ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]));
            System.out.println("The server is listening at: " +
                    serverSocket.getInetAddress() + " on port " +
                    serverSocket.getLocalPort());

            while (true) {
                Socket clientSocket = serverSocket.accept();
                MultiEchoServer mes = new MultiEchoServer(clientSocket);
                new Thread(mes).start();
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Multi-threaded Server (1/3)

```
public class MultiEchoServer implements Runnable {
   Socket clientSocket;

   public MultiEchoServer(Socket client) {
      clientSocket = client;
   }
```

Multi-threaded Server (2/3)

```
public void run() {
   try {
      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null) {
         System.out.println(inputLine);
         out.println(inputLine);
      }
   } catch (IOException e) {
      System.out.println(e.getMessage());
   }
}
```

Multi-threaded Server (3/3)

```
public static void main(String[] args) {
   if (args.length != 1) {
      System.err.println("Usage: java EchoServer <port number>");
      System.exit(1);
   }

   int portNumber = Integer.parseInt(args[0]);

   try {
      ServerSocket serverSocket = new ServerSocket(portNumber);
      System.out.println("The server is listening at: " +
            serverSocket.getInetAddress() + " on port " +
            serverSocket.getLocalPort());

      while (true) {
         Socket clientSocket = serverSocket.accept();
         MultiEchoServer mes = new MultiEchoServer(clientSocket);
         new Thread(mes).start();
      }
   } catch (IOException e) {
      System.out.println(e.getMessage());
   }
  }
}
```

Online Reference

- http://docs.oracle.com/javase/tutorial/essential/concurrency/

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

```
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;
    }
}
```

the c++ statement:

retrieve c
increment c
store value

the c-- statement:

retrieve c
decrement c
store value

Thread Interference 2

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

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

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

- Will only synchronize the block on `this`

Synchronized Blocks 2

```
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++; }
    }
}
```

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

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)