CPSC475
Thread Synchronization

(just scratching the surface)

Topics in This Section

- Overview - Sharing data between threads
- Synchronize keyword
- Volatile keyword
- Atomic Classes
- Concurrent Collections
- Useful methods

Overview - Sharing data between threads

- **A big deal- especially on multicore systems**
- **Threads can access/share any data created on heap**
  - If thread can find it that is

See http://www.javamex.com/tutorials/synchronization_synchronized_method.shtml
An excellent resource

What can’t a thread find?

- **Local Variables –** *are stored in each thread's own stack. That means that local variables are never shared between threads.*

```
public void someMethod(){

  long threadSafeInt = 0;

  threadSafeInt++;
}
```

What can’t a thread find?

- **Local Object References** – *are stored on heap but if another thread cannot get a ref to it you are OK.*

```
public void someMethod(){

  LocalObject localObject = new LocalObject();

  localObject.callMethod();
  method2(localObject);
}

public void method2(LocalObject localObject){
  localObject.setValue("value");
}
```

What can a thread find?

- **Object Members – that are stored on heap, and thread can get a reference to it. Its not** threadsafe

```
NotThreadSafe myNotThreadSafe;
private void notThreadSafeTask() {
    myNotThreadSafe = new  NotThreadSafe();

    //create 2 threads
    UpdateTask myTask1 = new UpdateTask();
    UpdateTask myTask2 = new UpdateTask();

    //start them they both have non threadsafe
    //access to myNotThreadSafe
    myTask1.execute();
    myTask2.execute();
}
private  class NotThreadSafe{
    StringBuilder builder = new StringBuilder();

    public void add(String text){
        this.builder.append(text);
    }
}

private class UpdateTask extends AsyncTask<Void, Void, Void
    :
```

Member var

Can get to myNotThreadSafe via Mainactivity.this.myNotThreadsafe.add

Not static has implicit access to enclosing object

Fix it - Synchronize keyword

- Like a traffic cop – 1 thread at a time
- Every Java ***OBJECT*** (no primitives) has an internal lock
  - So every object can be used for synchronization
- Issues- deadlock, complexity (others)
- Difficulty in debugging apps that 'almost' work

```
private  class ThreadSafe{
    private StringBuilder builder = new StringBuilder();

    public synchronized void add(String text){
        this.builder.append(text);
    }
}
```

Fix it - Synchronize Finer grain

```
public class Counter {
  private int count = 0;
  public void increment() {
    synchronized (this) {
      count++;
    }
  }
  public int getCount() {
    synchronized (this) {
      return count;
    }
  }
}
```

- **Every class has an associated lock**
- **Aquires lock and releases it when goes out of scope**
- **Other threads must wait until lock released**

See http://www.javamex.com/tutorials/synchronization_concurrency_synchronized1.shtml

Fix it - Synchronize Finer grain

- **Lock object**
  - Sometimes must synchronize across methods
  - Must explicitly unlock
    - Use try{}-finaly{} to guarantee unlock
    - difficult to ensure across methods

```
Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();
```

- **There is a lot to this topic**
- **See www.javamex.com**
  - It stops being scary after a few projects and starts to look interesting and challenging

Volatile keyword - Rationale

- **Compilers are free to move your code around to increase efficiency, speed as long as your code is not compromised**
- **Problem – In threaded application, rules change, how to tell compiler?**
- **When things go wrong you will have a hard time figuring it out. The code on the right will be the compiled equivalent of code on left**

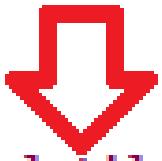
```
private boolean cancelled;
public void doVolatile(View v){
    cancelled= false;
    while(cancelled == false)//<==this is what
    {                        //   you want
        //do work
    }
}
```

Global

```
public void doVolatile1(View v){
    cancelled= false;
    while(true)      //<==this is what the
    {                //   compiler can do to you
        //you will never exit this
    }
}
```

Volatile keyword - Solution

- **Tells compiler that ‘variable's value will be modified by different threads’**
- **Do not reorder statements around it or optimize its value**
- **Store in main memory, not thread local storage**



```
private volatile boolean cancelled;
public void doVolatile(View v){
    cancelled= false;
    while(cancelled == false)//<==this is what
    {                        //   you want and
        //do work            //   what compiler
    }                        //   gives you
}
```

Atomic Classes

- **Objects, so ‘heavier’ than volatile. (So use volatile if you can)**
- **Handle cases where object needs synchronization for an operation that spans several steps**
- **CLASSIC: myInt++ is 3 separate operations**
  1. fetch myInt
  2. increment myInt
  3. write updated myInt back
- **Other thread access/modification between steps 1 and 3 is problematic**

Concurrent Collections

- **Need a HashMap, List or Queue for a threaded app?**
- **Its already there, written by experts, fast, scalable and best of all extensively tested.**
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

Useful Stuff

- **Logging threads**

```
Log.d(TAG, Thread.currentThread().getName() + " starting");
SystemClock.sleep(THREAD_WAIT_TIME);
Log.d(TAG, Thread.currentThread().getName() + " ending");
```

- **How do threads look in Android Studio?**
  - Show example

Summary

- **In a multithreaded concurrent world with multiple CPU cores, you will need to understand concurrency, threading and related issues.**
- **In Android you must move all time consuming processes off main thread.**
- **These slides just scratch the surface**

Reading

- **www.javamex.com**
- **Books ‘Java Concurrency in Practice’**

Of the 2, the web tutorial is much more approachable