Java Threads
Manuel Oriol - May 3rd, 2007
Thread

- a call to the method `start()` spawns a new thread (start a thread only once)
- The code executed is the one in `run()`
class SimpleThread extends Thread{
    static Integer a=0;
    int b;
    public SimpleThread(){
        synchronized(a){
            b=a++;
        }
    }
    public void run(){
        System.out.println(b);
    }
    public static void main(String []args){
        SimpleThread []threads = new SimpleThread[10];
        for (int i=0;i<10;i++){
            threads[i]=new SimpleThread();
        }
        for(SimpleThread myThread: threads ){
            myThread.start();
        }
    }
}
Runnable

- Implement the method run
- create a thread using the runnable object
ThreadGroup

• By default a thread belongs to its parent thread’s ThreadGroup

• These are a way of grouping threads and restricting accesses (protection domain)

• Hierarchical
Old Constructs???

- suspend
- resume
- destroy
- stop

Deprecated... really????

Throws NoSuchMethodError.

Throws:

NoSuchMethodError - always
So what?

- interrupt!

- or know what you are doing with locks...
Interrupt

**interrupt**

```java
public void interrupt()
```

Interrupts this thread.

Unless the current thread is interrupting itself, which is always permitted, the `checkAccess` method of this thread is invoked, which may cause a `SecurityException` to be thrown.

If this thread is blocked in an invocation of the `wait()`, `wait(long)`, or `wait(long, int)` methods of the `Object` class, or of the `join()`, `join(long)`, `join(long, int)`, `sleep(long)`, or `sleep(long, int)`, methods of this class, then its interrupt status will be cleared and it will receive an `InterruptedException`.

If this thread is blocked in an I/O operation upon an `interruptible channel` then the channel will be closed, the thread’s interrupt status will be set, and the thread will receive a `ClosedByInterruptException`.

If this thread is blocked in a `Selector` then the thread’s interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector’s `wakeup` method were invoked.

If none of the previous conditions hold then this thread’s interrupt status will be set.

**Throws:**

- `SecurityException` - if the current thread cannot modify this thread
Synchronizing with standard functionalities

• `join()`, `join(long)`
• `setPriority()` (higher is better)
• `yield()`
• `Thread.sleep(long)`
Volatile variables

- There is sequential consistency between the reads and writes of volatile variables
- Schematically:
  a read $r$ is allowed to observe a write $w$ to $v$ if $r$ does not happen before $w$ (and transitively)
Java Memory Model

- New in Java 5.0
- Fixes a significant number of problems... by asking changes in the JVMs

See reading from Manson, Pugh and Adve
Causality

- key to guarantee observable sequential consistency
- Intra-thread consistency
- synchronized actions are in mutual exclusion and respect causality
Example

```java
Class Reordering {
    int x = 0, y = 0;
    public void writer() {
        x = 1;
        y = 2;
    }

    public void reader() {
        int r1 = y;
        int r2 = x;
    }
}
```

Is it possible to have:

```
r1 == 2; r2 == 0?
```

Now?

No
Taking locks

- synchronized methods (static?)
- synchronized blocks
Example (1/3)

```java
public class ThreadExample {
    private static class MyThread extends Thread{
        private static int count;
        private static volatile int count2;
        private static int count3;
        Object lock;
        private MyThread(Object lock){
            this.lock=lock;
        }
    }
}
```
Example (2/3)

```java
public void run(){
    synchronized(lock){
        count++;
    }
    System.out.println("count="+count+" count2="+
        count2+" count3="+count3);
    count2++;
    System.out.println("count="+count+" count2="+
        count2+" count3="+count3);
    incCount();
    System.out.println("count="+count+" count2="+
        count2+" count3="+count3);
    synchronized static void incCount(){
        count3++;
    }
}
```
Example (3/3)

```java
public static void main(String[] args){
    Object o=new Object();
    for (int i=0; i<10; i++)
        (new MyThread(o)).start();
}
```
Example of run

count=1 count2=0 count3=0
count=1 count2=1 count3=0
count=1 count2=1 count3=1
count=2 count2=1 count3=1
count=3 count2=2 count3=1
count=4 count2=2 count3=2
count=7 count2=2 count3=2
count=8 count2=3 count3=2
count=8 count2=3 count3=3
count=9 count2=3 count3=3
count=10 count2=4 count3=3
Wait & Notify

- on an object
- wait releases the lock and then take it again
- notify notifies a waiter to try to grab the lock
- notifyAll notifies all the threads
Example

```java
synchronized(lock){
    lock.notifyAll();
    count++;
    try {
        lock.wait();
    } catch (InterruptedException e){}
}
```
Questions

• When do you need to have concurrency?
• When use a Thread, when use a Runnable?
• What is the difficulty with threads?
• How to ensure that you can use the deprecated methods?
Common Facts

• When having to wait on an input stream, one thread per input stream

• GUI use one thread, messing up with it is... not good.

• The Garbage Collector uses only one thread, messing up with it is... not good