Threads Fundamentals

A Thread is a separate execution task. Multiple Threads can execute in parallel. If you are on a single CPU processor, you might have one Thread waiting for I/O while another Thread is using the CPU.

There are many cases, where it is necessary to have 2 or more Threads executing in parallel.

- Streaming a Video
- Sending an email
- Downloading a file
- You are Comerica bank and you have ATM machines. To deal with the thousands of ATM machines, you might want to have a separate Thread to deal with each ATM machine. Of course each ATM machine must interact with the central bank.
- Server/Client relationships on the internet: web servers, our networked game projects.

There are 2 basic ways to create threads (A quick review from our Animation notes)

Approach1: Extend the Thread class

Key Points:

- Note that your programs always have at least one thread which starts running in main().
- Extending the Thread class ... take a look at online documentation to see this class
- Override the run method - This is where the Thread lives for it's whole life time.
- To make it run, construct it and call it's start method
- The sleep method is a useful capability for us now because we are interested in periodically doing something. You pass the number milleseconds that you want your Thread to sleep.
- Note that you have to catch the InterruptedException(checked Exception)

```
import java.util.Date;


public class MyThread1 extends Thread{

    public void run()
    {
        while (true)
        {
            try
            {
                Thread.sleep(1000); // Sleep for 1 second
                Date d = new Date();
                System.out.println("Date =" + d);
            }
            catch (InterruptedException e)
            {
            }
        }
    }
    public static void main(String[] args) {
        MyThread1 mt = new MyThread1();
        mt.start();
        System.out.println("Exitting main thread");

    }

}
```

Approach2: Implement the Runnable Interface

- A more flexible approach to using Threads is to use the Runnable Interface.

- In Java you can only extend one class. So the above Thread approach limits itself in how it can be used.
- Any class can implement the Runnable interface which has one required method: run
- To make a Thread, construct the class which implements Runnable, then Construct a Thread using the Thread constructor that has the pattern
  `Thread (Runnable r)`
- Then call the start method on the Thread constructed in the above step

```
import java.util.Date;


public class MyClass implements Runnable{

    public void run()
    {
        while (true)
        {
            try
            {
                Thread.sleep(1000); // Sleep for 1 second
                Date d = new Date();
                System.out.println("Date =" + d);
            }
            catch (InterruptedException e)
            {
            }
        }
    }
    public static void main(String[] args) {
        MyClass mc = new MyClass();
        Thread t = new Thread(mc); // This constructor needs any class which implements Runnable
        t.start();

        System.out.println("Exitting main thread");

    }

}
```

Managing Threads

You might have noticed in the above example, that we have a "Delinquent Mother child relationship". The main thread starts up in main() and launches a MyThread1 thread. Then main exits which means that this thread is gone. Meanwhile the child thread runs forever with no other thread telling it to quit. Of course in Eclipse we can hit the red box to terminate everything.

To avoid offensive thoughts of Mother threads killing or terminating children threads, I will try to use "Controlling Thread" and "Servant Thread" terminology. However, I may slip up and talk about Mother/Children relationships because I have heard and used this terminology quite a bit before. Please forgive me when I do.

- In the following we will launch 3 Servant Threads from main. Each Servant thread will be passed a unique "Sleep Time" and use this to identify itself when it runs.
- The main thread (our main() code) will start up the three Servant Threads and not go away until all of the Servant Threads have terminated.
- The user is prompted for which of the 3 Servant Threads to terminate. The Main thread then calls the **`"interrupt"`** method to terminate the Servant Thread (see the killThread routine).
- After the "interrupt" call on a Thread, the Servant Thread will experience an "InterruptedException". At this point the Servant thread voluntarily exits it's run method.
- Note that Servant thread does not go away immediately. It has to be scheduled and execute its part of the termination sequence. Because of this, we have the Main Thread sleep for 1 second to give the Servant thread a chance to go away.
- After this 1 second sleep, the Main Thread will call **`"anyThreadsStillRunning"`** to check to see if there are any Servant threads still running. Note that there is a key routine in the Thread class called **`isAlive()`** that can tell you whether a thread is still running.

- If no Servant threads are still running, the Main Thread exits.

MyThread2.java

```
import java.util.Date;
import java.util.Scanner;


public class MyThread2 extends Thread{

    int sleepTime;
    MyThread2(int sleepTime)
    {
        this.sleepTime = sleepTime;
        System.out.println("Constructing MyThread2("+sleepTime +")");
    }

    public void run()
    {
        boolean continueFlag = true;

        while (continueFlag)
        {
            try
            {
                Thread.sleep(sleepTime); // Sleep
                Date d = new Date();
                System.out.println(sleepTime + " Thread wakes up at Date =" + d);
            }
            catch (InterruptedException e)
            {
                System.out.println(sleepTime + " was just interrupted");
                continueFlag = false;
            }
        }
    }

    static boolean anyThreadsStillRunning(Thread[] myThreads)
    {
        for (int i=0; i < myThreads.length; i++)
        {
            if (myThreads[i].isAlive())
                return true;
        }
        return false;
    }
    static void killThread(Thread t)
    {
        t.interrupt();
        // We want to pause to give the thread a chance to terminate
        // before checking which threads are still running.
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Dont expect to get here ... EVER");
        }
    }

    public static void main(String[] args) {
        int[] sleepTimes = {5000, 8000, 10000};
        Thread[] myThreads = new Thread[sleepTimes.length];

        // Start up my Threads
```

```
        for (int i=0; i < sleepTimes.length; i++)
        {
            myThreads[i] = new MyThread2(sleepTimes[i]);
            myThreads[i].start();
        }

        Scanner keyboard = new Scanner(System.in);
        do
        {
            System.out.println("Enter the index of which thread you want to kill (0, 1, 2)");
            int killIndex = keyboard.nextInt();
            killThread(myThreads[killIndex]);

        } while(anyThreadsStillRunning(myThreads));

        System.out.println("Exitting main thread");

    }

}
```

Last Updated: March 28, 2014 1:18 PM ---