Fundamentals of Java

Threads

Threads

A thread is a sequence of executable instructions.

instruction1;

instruction2

instruction3;

...

instructionN;

Threads

Java allows multiple threads to be executed in parallel. On a single-core CPU, the appearance of concurrency is achieved with timesharing:

Thread #1	**Thread #2**
instruction1; instruction2; instruction3; instruction4; instruction5;	 instruction1; instruction2; instruction3; instruction4; instruction5;

Threads

On a multi-core CPU, two threads can be genuinely be executed in parallel:

Thread #1	**Thread #2**
instruction1; instruction2; instruction3; instruction4; instruction5;	instruction1; instruction2; instruction3; instruction4; instruction5;

Threads

In a standard Java application, a "main" thread is automatically created for you, and your "main" method is executed within this thread.

If you wish to take advantage of multi-core CPUs, you must create additional threads.

Even with single-core CPUs, multi-threaded algorithms can often be more efficient than single-threaded algorithms!

Java classes for threading

Package java.lang provides the following interface representing a sequence of instructions that can be "run":

```
public interface Runnable
{
   public void run();
}
```

Runnables

You can define a runnable by implementing this interface:

```
public class CalculateNthPrime implements Runnable {
    private int n;
    public CalculateNthPrime(int n) { this.n = n; }
    public void run() { // only works if n > 2
        int i = 3, count = 2, prime = 2;
        while (count < n)
            if (isPrime(i)) { count++; prime = i; i += 2; }
        System.out.println("Nth prime is " + prime);
    }
}
```

Creating a Thread

Once you have a Runnable, you can start it in a new Thread (also from package java.lang):

```
Runnable rbl = new CalculateNthPrime(1000);
Thread t = new Thread(rbl);
t.start(); // Non blocking
System.out.println("Thread started");
```

The start() method executes the thread in the background. "Thread started" will be printed immediately.

Subclassing Thread

Instead of creating a Runnable, it is also possible to subclass Thread:

```
public class NthPrimeThread extends Thread {
    private int n;
    public NthPrimeThread(int n) { this.n = n; }
    public void run() { // only works if n > 2
        int i = 3, count = 2, prime = 2;
        while (count < n)
            if (isPrime(i)) { count++; prime = i; i += 2; }
        System.out.println("Nth prime is " + prime);
    }
}
```

Subclassing Thread

Create and start the thread as follows:

```
NthPrimeThread thread = new NthPrimeThread(1000);
thread.start(); // non-blocking
System.out.println("Thread started");

thread.join(); // blocking
System.out.println("Thread finished");
```

Use thread.join() to wait for a thread to finish.

Threads vs Processes

A process represents a running program.

Each process has separate memory.

Multiple threads can run within a single process.

Each thread in a process shares the same memory.

Shared memory allows for efficient communication/cooperation between threads (with some pitfalls...)

Dangers of Multithreading

```
public class CounterThread extends Thread {
    public static int counter = 0;
    public void run() {
        for (int i = 0; i < 10000; i++)
            counter++;
    }
}
...
CounterThread thread1 = new CounterThread();
CounterThread thread2 = new CounterThread();
thread1.start(); thread2.start();
thread1.join(); thread2.join();
System.out.println("Counter is " + CounterThread.counter);
```

WHAT WILL THIS PRINT?

Dangers of Multithreading

The line count++ is really three steps:

1. retrieve the current value of counter
2. compute current value + 1
3. store the new value into counter

What happens if two threads attempt to execute this procedure at roughly the same time, on shared memory?

Dangers of Multithreading

Thread #1	**Thread #2**
(counter is currently 25) retrieve the current value -> 25 compute 25 + 1 -> 26 store 26	(counter is currently 25) retrieve the current value -> 25 compute 25 + 1 -> 26 store 26

We *expect* the result to be 27 since the procedure is executed twice.

However, the two threads are interfering with each other.

Solution: synchronisation (In Java: "synchronization")

The following three steps are a *critical section*:

1. retrieve the current value of counter
2. compute currentValue + 1
3. store the new value

While this code is executing, no other thread should be allowed to manipulate the counter at the same time.

The "synchronized" keyword

Synchronisation is achieved using the "synchronized" keyword:

```
public class CounterThread extends Thread {
    public static int counter = 0;
    private static Object monitor = new Object();
    public void run() {
        for (int i = 0; i < 10000; i++) {
            synchronized (monitor) {
                counter++;
            }
        }
    }
}
```

The "synchronized" keyword

```
synchronized (obj) {
    counter++;
}
```

The parameter to synchronized() is called the lock or the "monitor". It can be any object.

Whenever a thread enters synchronized(**obj**) block, it obtains a lock on the obj monitor. Upon existing the block, the lock is released.

While holding the lock, no other thread can enter a block that requires that lock.

Dangers of Locking

Thread #1	Thread #2
synchronized (lock1) { **synchronized (lock2) {**	synchronized (lock2) { **synchronized (lock1) {**

At this point, thread #1 is waiting to acquire lock2.

And.... thread #2 is waiting to acquire lock1.

This is called DEADLOCK.