5COSC019W - OBJECT ORIENTED PROGRAMMING
Lecture 8: An Introduction to Java Threads Part 1

Dr Dimitris C. Dracopoulos

Why Threads?

Time sharing makes a computer system give the impression that several things are executed simultaneously.
Unlike OS processes, threads:

- Are much more lightweight: they do not have the costly overhead of processes (saving the program state, memory contents, file descriptors, etc.)
- Use the same address space.

Applications of Threads

Threads allow you to do many things at the same time. Typical cases that this is desirable are:

- Interactive programs that never look “dead” to the user. For example, one thread might be controlling and responding to a GUI, while another is making the computations requested and a third thread is doing file I/O, all for the same program.
- Some programs are easier to write if you split them into threads.

  For example in a client server system, the server can spawn a new thread for each new request. This thread will be responsible to fulfill each each request. Contrast this with the alternative where a single threaded server has to keep track of the state of each individual request.
- Some programs are amenable to parallel processing. Examples: some sorting algorithms, matrix operations, etc.

How to create a Java Thread

Two ways:

1. Extend the class `Thread` and implement the `run()` method :

```
class Worker extends Thread {
    public void run() { ... }
}

Worker w = new Worker();
w.start();
```

2. Implement the `Runnable` interface (the class `Thread` is an implementation of it):

```
class Sort implements Runnable {
    public void run() { ... }
}

Sort s = new Sort();
Thread t1 = new Thread(s);
t1.start();
```

Characteristics of the Runnable interface

If you implement a thread using the Runnable interface:

- You can only call the start() method of the Thread.
- You cannot explicitly call the other methods (e.g. sleep()) provided by class Thread:

```
public class example {
   public static void main(String[] a) {

      // alternative 1
       ExtnOfThread t1 = new ExtnOfThread();
       t1.start();

      // alternative 2
       Thread t2 = new Thread (new ImplOfRunnable());
       t2.start();
   }
}
```

Characteristics of the Runnable interface (cont'ed)

```
class ExtnOfThread extends Thread {
   public void run() {
       System.out.println("Extension of Thread running");
       try {sleep(1000);}
       catch (InterruptedException ie) {return;}
   }
}

class ImplOfRunnable implements Runnable {
   public void run() {
       System.out.println("Implementation of Runnable running");

       // next two lines will not compile
       // try {sleep(1000);}
       // catch (InterruptedException ie) {return;}
   }
}
```

Characteristics of the Runnable interface (cont'ed)

- To do this, you must first create the thread object associated with Runnable:

```
class ImplOfRunnable implements Runnable {
    public void run() {
      System.out.println("Implementation of Runnable" +
                         "running");

        Thread t = Thread.currentThread();

        try {t.sleep(1000);}
        catch (InterruptedException ie) {return;}
    }
}
```

Thread Priorities and the Lifecycle of a Thread

- Threads have priorities.
- A thread with a higher priority will pre-empt another one with lower priority. This means that if a thread with priority x is running, then it will be pushed off the processor by another thread with priority $y > x$.
- The developer can set the priority of a thread, since the only thing he affects is his own program.
- Time slicing does not happen necessarily. This depends on the OS scheduling algorithm, and effectively it means that a running thread might not share the processor with threads of equal priority.
- To ensure that other threads with equal priority will be given the chance to run, `yield()` could be called periodically in your thread. The scheduler is free to ignore this hint.

Example of Thread Creation

```java
class Coffee extends Thread {
    public void run() {
        while(true)
            System.out.println("I like coffee");
    }
}

class Tea extends Thread {
    public void run() {
        while(true)
            System.out.println("I like tea");

    }
}

public class drinks {
    public static void main(String[] a) {
        Coffee t1 = new Coffee();
        t1.start();
        new Tea().start();  // an anonymous thread
    }
}
```

Output:

```
I like coffee
I like coffee
I like tea
I like coffee
I like coffee
I like coffee
I like tea
I like coffee
I like coffee
I like coffee
I like coffee
I like coffee
I like coffee
I like coffee
I like tea
I like coffee
```

Categories of Thread Programming

Four levels of difficulty based on the type of synchronisation needed between threads:

- Unrelated Threads.
- Related but Unsynchronised Threads.
- Mutually-Exclusive Threads.
- Communicating Mutually-Exclusive Threads.

Related but Unsynchronised Threads

An example is a client server system where the server spawns a new thread for each client request.

This type can also be used to partition a large data set into smaller subsets on which a different thread operates. For example to find whether a given large number is prime one can split the range of numbers searched and give the relevant interval to a particular thread :

```java
class testRange extends Thread {
    static long possPrime;
    long from, to;

    /* args: start of range of factors attempted and number to test */
    testRange(int argFrom, long argpossPrime) {
        possPrime = argpossPrime;
        if (argFrom == 0)
            from = 2;
        else
            from = argFrom;

        to = argFrom + 99;
    }

    public void run() {
        for (long i=from; i<=to && i<possPrime; i++)
            if (possPrime % i == 0) {  // i divides possPrime exactly
                System.out.println(
                     "factor " + i + " found by thread " + getName());
                break;  // get out of for loop! - found a prime
            }
    }
}
```

```java
public class testPrime {
    public static void main(String s[]) {
        if (s.length!=1) {
            System.out.println("usage: java testPrime somenumber");
            System.exit(0);
        }
        long possPrime = Long.parseLong(s[0]);
        int centuries = (int)(possPrime/100) +1;

        for(int i=0;i<centuries;i++) {
             new testRange(i*100, possPrime).start();
        }
    }
}
```

Mutually Exclusive Threads

Threads operate on the same address space. We should make sure that they do not try to modify simultaneously the same piece of data (this is known as *mutual exclusion*).

Example: Simulate a steam boiler. Define the current reading of the pressure and the safe limit for a pressure gauge. Create a number of threads, each one of them looking at the current reading and if it is within the allowed limits, it tries to increase the pressure.

```java
public class p {
   static int pressureGauge=0;
   static final int safetyLimit = 20;

   public static void main(String[]args) {
      pressure[] p1 = new pressure[10];
      for (int i=0; i<10; i++)  {
         p1[i] = new pressure();
         p1[i].start();
      }
      try {
         for (int i=0; i<10; i++)
         p1[i].join();
      } catch(Exception e){ }

     System.out.println("gauge reads "
        + pressureGauge + ", safe limit is 20");
   }
}
```

```
// Now let's look at the pressure thread.
// This code simply checks if the current pressure reading
// is within safety limits, and if it is, it waits briefly,
// then increases the pressure. Here is the thread:

class pressure extends Thread {
    void RaisePressure() {
        if (p.pressureGauge < p.safetyLimit-15) {
           // wait briefly to simulate some calculations
           try{sleep(100);} catch (Exception e){}
           p.pressureGauge += 15;
        }  else ; // pressure too high -- don't add to it.
    }

    public void run() {
        RaisePressure();
    }
}
```

The output when the program runs is:

```
gauge reads 150, safe limit is 20
```

The “synchronized” keyword

To achieve mutual exclusion oven an entire class then a `static` method of a class must be defined as `synchronized`:

```
public static synchronized void RaisePressure()
```

This means, that at any point in time only one thread can execute this method and all other threads are waiting.

- To achieve this, the thread which executes this method, obtains the “lock” of the whole class.
- There can be many threads executing the various parts of the same class at the same time, but only one thread executing the static synchronised method. E.g. there can be many threads executing method `foo()` of a class at the same time but only one Thread executing the method `RaisePressure()`.

Synchronising non-static methods

One can define a non-static method to be `synchronized`.

- This implies that a thread which executes a non-static synchronized method, will obtain the lock for the actual object and not the entire class.
- When the thread having the lock of the object finishes the execution of the synchronised method, it releases the lock and another thread can obtain it and execute the synchronized method.
- Again multiple threads can execute the same method of a class as long as this is not synchronised. However at any point in time, only one thread can execute any method defined as synchronised. If there are 2 methods defined as synchronised then at any point in time, only one thread can access any portion of the synchronised code.

Synchronising non-static methods (cont'ed)

Note that in the gauge example making the following change:

```
synchronized void RaisePressure() {
   ...
}
```

will not achieve mutual exclusion. This is because each of the 10 different threads will obtain the `pressure` object's "lock". This is different in each thread.

The correct approach is to modify `RaisePressure` as:

```
// alternative, safe version!
static synchronised void RaisePressure() {
    if (p.pressureGauge < p.safetyLimit-15) {
         try {
            sleep(100);  // delay
         }
         catch (Exception e) {
         }

         p.pressureGauge += 15;
     }
}
```

Synchronising a portion of code

A portion of code can be synchronised by acquiring the lock of any object. Then threads which require the the lock of the same object will have to wait until the thread which has the lock releases it:

```
synchronized(O) {
   ....// some code
}
```

where O is any object. Only one thread will be able to execute the code enclosed in a `synchronized` block at any point in time. Therefore:

```
synchronized void foo() { ... }
```

is equivalent to:

```
void foo() {
   synchronized(this) {
      ...
   }
}
```

More on Synchronisation

Synchronisation solves 2 problems:

- *Thread interference* (seen in the previous boiler example)
- *Memory consistency*: Modifications to data by a thread are not visible (in time) by another thread.

If two or more threads attempt to update the same value simultaneously a race condition can occur!

$\Longrightarrow$ **A race condition occurs if the result of multiple threads on shared data depends on the order in which the threads are scheduled!**

Another example of Thread Interference

Thread interference can occur even if an operation is applied in a single simple statement:

```java
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

Another example of Thread Interference (cont'ed)

The bytecode that the virtual machine executes for each of the above single statements (e.g. `c++`) could be composed by 3 steps:

1. Retrieve the current value of `c`
2. Increment the retrieved value by 1
3. Store the incremented value back in `c`

Thus, `interleaving` can occur even in this case. Two threads, one executing `c++` and the second `c--` could result in a value of -1 or other value.

Communicating Mutually Exclusive Threads

Synchronisation via the `wait()`, `notify()` calls.

- The `wait()` and `notify()` are always called from within synchronised code.

1. The executing thread may notice that some data isn't ready for it yet, and `wait()` for it.
    1.1 It executes `wait()` and goes to the wait list.
    1.2 It releases the lock, allowing one thread to proceed from the blocked (ready list). Threads on the blocked list are blocked until they can get the synchronisation object, but otherwise they are ready to run.
2. Eventually one thread will produce some data, then it will call `notify()` waking a thread on the wait list, moving it to the blocked (ready) list. If more than one threads exist in the wait list, then the thread which will wake up will be chosen randomly.
3. When the thread that just called `notify()` leaves the method, it gives something else the chance to run.

Communicating Mutually Exclusive Threads (cont'ed)

At any point in time there are:

- One running thread
- A number of threads waiting in the wait list. Threads which execute `wait()` are the ones which are moved to this list.
- A number of threads waiting in the blocked (ready) list. These are threads which are ready to run but wait for the lock of the synchronisation object to be released.

The Producer - Consumer Example

Communication is achieved by the following code:

```
// producer thread - assumes no limits in storing produced data
enter synchronized code (i.e. grab mutex lock)
    produce_data();
    notify();
leave synchronized code (i.e. release lock)


// consumer thread
enter synchronized code
    while (no_data)
       wait();
    consume_the_data();
leave synchronized code
```

```
// producer thread produces only one datum in a bounded buffer
enter synchronized code (i.e. grab mutex lock)
while (buffer_full)
   wait();
produce_data();
notify();
leave synchronized code (i.e. release lock)


// consumer thread consumes one datum
enter synchronized code
while (no_data)
   wait();
consume_the_data();
notify();
leave synchronized code
```

The full Producer Consumer Example

```java
public class plum {
  public static void main(String args[]) {
     Producer p = new Producer();
     p.start();

     Consumer c = new Consumer(p);
     c.start();
  }
}

class Producer extends Thread {
    private String [] buffer = new String [8];
    private int pi = 0;  // produce index
    private int gi = 0;  // get index

    public void run() {
       // just keep producing
       for(;;) produce();
    }
```

The full Producer Consumer Example (cont'ed)

```
private final long start = System.currentTimeMillis();
private final String banana() {
    return "" + (int) (System.currentTimeMillis() - start);
}

synchronized void produce() {
   // while there isn't room in the buffer
   while ( pi-gi+1 > buffer.length ) {
       try {wait();} catch(Exception e) {}
   }
   buffer[pi%8] = banana();
   System.out.println("produced["+(pi%8)+"] "+buffer[pi%8]);
   pi++;
   notifyAll();
}
```

The full Producer Consumer Example (cont'ed)

```
    synchronized String consume(){
       // while there's nothing left to take from the buffer
       while (pi==gi) {
           try {wait();} catch(Exception e) {}
       }
       notifyAll();
       return buffer[ gi++ % 8 ];
    }
}
```

The full Producer Consumer Example (cont'ed)

```
class Consumer extends Thread {
   Producer whoIamTalkingTo;
   // java idiom for a constructor
   Consumer(Producer who) {
       whoIamTalkingTo = who;
   }

   public void run() {
      java.util.Random r = new java.util.Random();
      for(;;) {
          String result = whoIamTalkingTo.consume();
          System.out.println("consumed: "+result);
          // next line is just to make it run a bit slower.
          int randomtime = Math.abs(r.nextInt() % 250);
          try{sleep(randomtime);} catch(Exception e){}
      }
   }
}
```

The output of this program is:

```
produced[0] 1
consumed: 1
produced[1] 8
produced[2] 9
produced[3] 10
produced[4] 10
produced[5] 10
produced[6] 10
produced[7] 10
produced[0] 11
consumed: 8
produced[1] 198
consumed: 9
produced[2] 318
consumed: 10
produced[3] 338
consumed: 10
```