

APIT: Concurrency

Simon Rogers

13/01/2019

Overview

- ▶ Concurrency
- ▶ Threading
- ▶ Solving threading problems
- ▶ Threads in Swing

What is concurrency?

- ▶ Multiple parts of the program running simultaneously
- ▶ Why?
 - ▶ Make use of multiple processors
 - ▶ Efficient integration with slow devices (e.g. disks)
 - ▶ User-friendly-ness (responsive OS)
- ▶ Useful (but oldish) paper
- ▶ In Java, we can use `Threads` to build concurrent programs

Mental models

- ▶ Previously you might have had the model in your head of the computer **being** somewhere in your programme.
 - ▶ single *point of execution*
- ▶ With multiple *threads*, each thread is (potentially) in a different place *at the same time!*
 - ▶ multiple *points of execution*

Motivating Example

- ▶ Making a GUI programme that will count down (counting down every 5th of a second) a number entered into a JTextField when a button is pressed
- ▶ BadFrame.java: obvious solution doesn't work
- ▶ GoodFrame.java: needs threads.
 - ▶ In this case, things are complicated by the use of Swing to make the GUI.
 - ▶ We will start with general Java threads and return to Swing later

Threads in Java are objects

- ▶ Our previous objects have all been passive
- ▶ Thread objects are active:
 - ▶ They have attributes and methods
 - ▶ And each one has its own *point of execution* - i.e. they start running by themselves

Creating threads in Java

There are two ways of creating threads in Java:

- ▶ You must create a class that either:
 - ▶ Implements the `Runnable` interface
 - ▶ Extends the `Thread` class

Implementing the Runnable interface

The Runnable interface is very simple:

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

To use it:

- ▶ create a new class implementing this interface
- ▶ create a Thread object passing an instance of your class
- ▶ call Thread.start() (**Note: we never call run()**)
- ▶ our new class *must* have a run() method.

To use this class, we must create an instance and then place this instance within an instance of the Thread class:

```
public class RunnableTest {  
    public static void main(String[] args) {  
        PointlessPrint p = new PointLessPrint("Hello",100);  
        Thread t = new Thread(p);  
        t.start();  
    }  
}
```

t.start() starts the thread by invoking the run() method of PointlessPrint

Creating many threads is straightforward via an array of Thread objects:

```
public static void main(String[] args) {  
    int nThreads = 2;  
    Thread[] threads = new Thread[nThreads];  
    for(int i=0;i<nThreads;i++)  
    {  
        PointlessPrint p = new PointlessPrint(  
            "I am thread " + i,10);  
        threads[i] = new Thread(p);  
        threads[i].start();  
    }  
}
```

Producing the following output:

```
0/10 I am thread 0
1/10 I am thread 0
0/10 I am thread 1
1/10 I am thread 1
2/10 I am thread 0
2/10 I am thread 1
.
.
6/10 I am thread 1
7/10 I am thread 0
7/10 I am thread 1
8/10 I am thread 0
8/10 I am thread 1
9/10 I am thread 0
9/10 I am thread 1
```

We can see from the order of the `println` statements that both threads are running at the same time.

- ▶ The order might change every time we run it
- ▶ The program stops once all threads are complete
- ▶ It's impossible for us to know when Java switches from one thread to another
- ▶ Note: They're not necessarily on different processors / cores, but might be

Extending the Thread class

- ▶ The alternative to implementing the `Runnable` interface
- ▶ Create a new class that extends `Thread`
- ▶ The new class has to have a method that overrides `run()`
- ▶ The equivalent to our previous example can be found in `SimpleThreadTest`

Extending v Implementing

- ▶ Extending Thread might seem appealing as it saves some lines
- ▶ But remember - classes can only extend one other class
- ▶ Often, implementing Runnable will be more sensible

Have you been paying attention?!

Can you predict the output of this?

```
public class MainThread extends Thread{
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
        System.out.println("Thread finished");
    }
    public static void main(String[] args) {
        for(int i=0;i<10;i++) { new MainThread().start(); }
        System.out.println("THE END");
    }
}
```


Thread names

- ▶ In our examples, we passed a message to a thread to help identify it
- ▶ Threads can also be given names through their constructor:

```
Thread t = new Thread(aRunnableThing, "my name");
```

```
Thread t = new Thread("my name");
```

- ▶ which can be accessed via:

```
thread.getName()
```

- ▶ See notes for full examples
- ▶ But...within a class that extends Thread: use `this.getName()` (or just `getName()`)
- ▶ Within a class that extends Runnable you have to obtain the relevant Thread object with `Thread.currentThread()`
- ▶ So: `Thread.currentThread().getName()`

Blocking methods

- ▶ Blocking methods are methods that rely on something else within the system for termination
 - ▶ Waiting for a timer to elapse
 - ▶ Waiting for another thread to end
- ▶ Because these methods rely on something external, they might be waiting forever.
- ▶ To ensure smooth running, they have to be *cancelable*

Interrupted Exception

- ▶ Threads can be interrupted by other threads
- ▶ When a thread is interrupted, one of two things happen:
 - ▶ If it is running an interruptable method (e.g. `Thread.sleep()`), the method unblocks and throws the `InterruptedException`
 - ▶ Otherwise, its (boolean) interrupted status is set
- ▶ Interrupted status can be read with `Thread.isInterrupted()`
- ▶ Interrupted status can be read and reset (0) with `Thread.interrupted()`

Sleeping threads

- ▶ `Thread.sleep(long time)` is a blocking method that can be stopped by interrupting. If this happens, it throws `InterruptedException`
- ▶ `InterruptedException` has to be caught:

```
public void run() {  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        System.out.println("You woke me up");  
    }  
}
```

Join

- ▶ In many applications it will be useful to know if a Thread has finished.
- ▶ `someThread.join()` pauses the current thread until `someThread` has finished.
- ▶ `join()` throws an `InterruptedException`
- ▶ Syntax: `aThread.join()` pauses the thread **that calls the method** until `aThread` has finished.
 - ▶ This can be a bit confusing!

The following code starts 5 threads and then waits for each in turn to stop:

```
MyThread[] m = new MyThread[5];  
for(int i=0;i<5;i++) {  
    m[i] = new MyThread();  
    m[i].start();  
}  
for(int i=0;i<5;i++) {  
    m[i].join();  
}
```

main doesn't finish until after the last thread

Note: the following is not good:

```
MyThread[] m = new MyThread[5];  
for(int i=0;i<5;i++) {  
    m[i] = new MyThread();  
    m[i].start();  
    m[i].join();  
}
```

Why?

The benefits of parallel processing

- ▶ Many machines have multiple cores / processors
- ▶ Threads can be placed on different cores / processors
 - ▶ Java does this for us - we have no control
- ▶ Running things in parallel should give us speed improvements
 - ▶ Although it increases system book-keeping
- ▶ Class example: merge sort

Merge sort

- ▶ We'd like to sort the values in a large array.
- ▶ Can be made parallel:
 - ▶ Split the array into N smaller arrays
 - ▶ Sort the smaller arrays
 - ▶ Merge the results together
- ▶ How much speed-up will this give?

Shared variables

- ▶ A benefit of threads is the shared address space
 - ▶ Multiple threads can access the same shared resources
- ▶ For example, suppose I would like to make a system where several threads can all increment the same counter
- ▶ See CounterExample
- ▶ Why can't we just pass an Integer around instead of a MyCounter object? (see Immutable objects and Immutable objects 2)

- ▶ If we have many threads accessing the same shared object we don't always see what we might expect.
- ▶ In this example, if we have 100 threads all incrementing the same counter 1000 times then we should see 100000 at the end.
- ▶ But we don't. . . any ideas why not?

```
public void run() {  
    for(int i=0;i<n;i++) {  
        int temp = count.getCount();  
        temp++;  
        count.setCount(temp);  
    }  
}
```

The problem is found in the run() method:

- ▶ Remember that we have no idea when Java will move from one Thread to another.
- ▶ If it moves between the getCount() and setCount() methods...

Mycounter.count

Thread 1

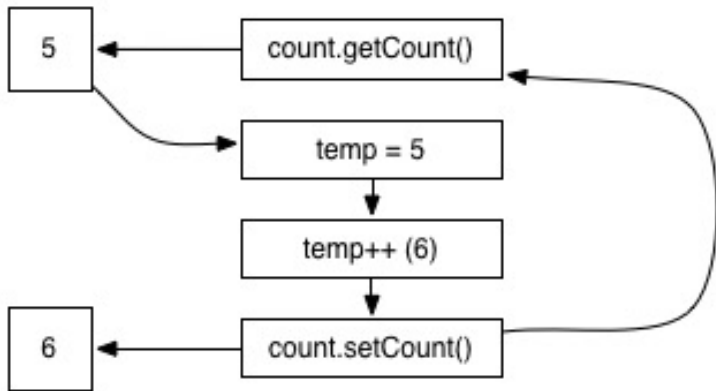


Figure 1: Single thread operation

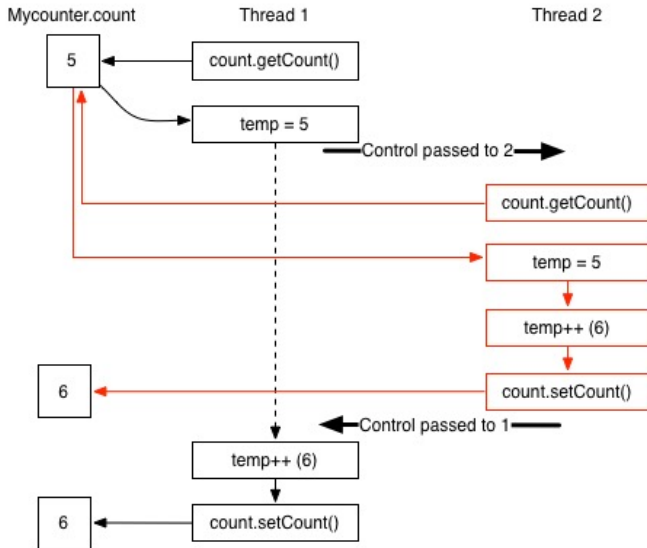


Figure 2: Multi-thread operation

- ▶ If control is passed between get and set the new thread sees an out of date value.
 - ▶ Remember: `temp` is local to each thread.
- ▶ In reality, thread 1 might be sitting dormant with `temp=5` for a long time
- ▶ When it finally updates it, it will effectively delete lots of updates performed by other threads
- ▶ It is known as a *race condition*
- ▶ This is a *big* problem in multi-threaded programs
- ▶ We'll now look at ways of overcoming it

Monitors

- ▶ Before we solve this problem, we need to know about *monitors*
- ▶ Every object has a *monitor*
- ▶ *monitors* help us to synchronise Threads
- ▶ If one Thread locks the monitor on a particular object, other Threads cannot.
- ▶ Other Threads *wait* until the original Thread unlocks

The life-cycle of a Thread

- ▶ In the previous slide we mentioned Threads *waiting*
- ▶ Threads can exist in various states
 - ▶ We need to understand the *blocked* and *waiting* states
 - ▶ We will refer back to the following diagram later. . .

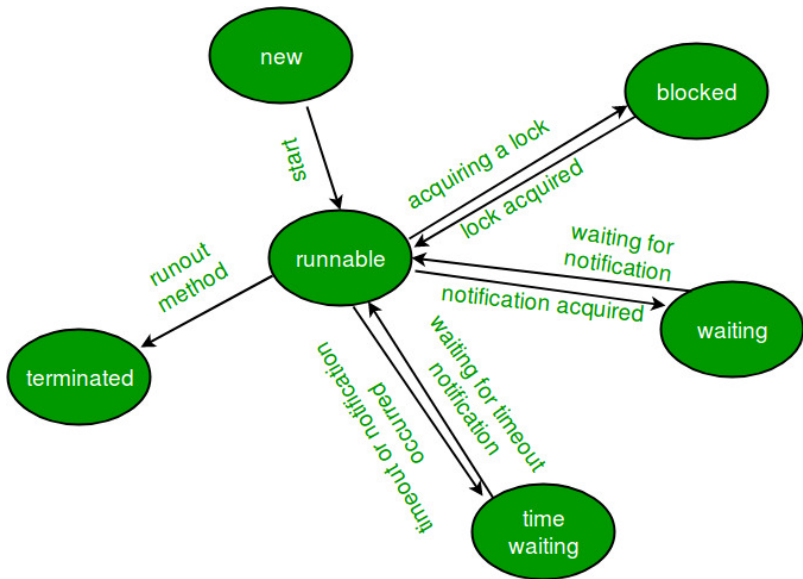


Figure 3: Thread states - Source: Core Java Vol 1, 9th Edition, Horstmann, Cay S. & Cornell, Gary_2013

Synchronized

- ▶ To overcome race conditions, we must *lock* the monitor of the shared object.
- ▶ Other threads trying are *blocked* until the lock is released
- ▶ The easiest way is with `synchronized` blocks and methods.

- ▶ CounterExample3 gives a new version of our Counter program
- ▶ Note that the incrementation is now done inside the class
- ▶ It still has the same problem, although perhaps not as extreme (try it and see. Why? Think about how `count++` is done and the chances of being interrupted at a bad point)
- ▶ We can solve our race condition by making the `increment()` a synchronised method
 - ▶ when any thread is invoking a synchronized method, all other threads trying to invoke it are paused until it has finished
- ▶ See CounterExample4 ... problem solved

- ▶ Alternatively, we can just synchronize a block of code.
- ▶ E.g. instead of declaring `increment()` as synchronized we can:

```
public void run() {  
    for(int i=0;i<n;i++) {  
        synchronized(count) {  
            count.increment()  
        }  
    }  
}
```

- ▶ This causes the thread to lock the count object
- ▶ No other threads can modify count when the thread is in this block.
- ▶ This will also fix the problem - try it
- ▶ There are other blocks that could be synchronized - try some
- ▶ Threads that get to the synchronized code when it is locked enter the *blocked* state (see previous diagram)

wait and notify

- ▶ In more complex examples, we might need a one Thread to tell another when something has happened.
- ▶ wait and notify (and notifyAll) allow us to do this
- ▶ They work with respect to a particular Object

- ▶ The process:
 - ▶ A Thread synchronizes an object and then called `Object.wait()`
 - ▶ it is then moved to the *waiting* state
 - ▶ Other Threads can now synchronize the object (and maybe also enter the *waiting* state)
 - ▶ If another Thread synchronizes the Object and then calls `Object.notify()` one waiting Thread is moved from *waiting* to *blocked* and will continue once it can regain synchronization

A simple example – Waiter and Notifier

- ▶ We will create an example with two Threads and one shared object
- ▶ The type of object is irrelevant
- ▶ One Thread will synchronise and then wait
- ▶ The other will sleep a while then synchronize and notify
- ▶ At this point, the other Thread will be awoken and can continue *once* the notifier relinquishes the monitor (i.e. leaves the synchronized block)
- ▶ `Waiter.java`, `Notifier.java`, `WaiterNotifier.java`
- ▶ The difference between `notify` and `notifyAll` will become apparent if there are multiple Waiter objects

Locks

- ▶ An alternative approach involves creating Lock objects
- ▶ For example, ReentrantLock() (CounterExample5):

```
public static class MyCounter {  
    private int count = 0;  
    private ReentrantLock counterLock =  
        new ReentrantLock();  
    public void increment() {  
        counterLock.lock();  
        count ++;  
        counterLock.unlock();  
    }  
    ...  
}
```

- ▶ When counterLock is locked, no other thread can lock it until it has been unlocked

- ▶ There's a problem: if the code between `lock` and `unlock` throws an exception the `unlock` never happens
- ▶ Always do the following to ensure the lock is released:

```
someLock.lock();  
try {  
    // Some code  
}  
finally {  
    someLock.unlock();  
}
```

Deadlocks

- ▶ What if two threads are both waiting for one another to release a lock?
- ▶ The program will hang indefinitely
- ▶ This is a *deadlock*
- ▶ For example, suppose adding another object to our CounterExample that decrements MyCounter
 - ▶ If we set the system up so that in total the same amount is incremented and decremented then count might sometimes become negative (depending on ordering of events)
 - ▶ See CounterDecounter

- ▶ We would like to ensure this number never goes negative
- ▶ One way of doing this would be to put some kind of wait condition in the decrement method (CounterDecounter2):

```
counterLock.lock();
try {
    while(count<amount) {
        Thread.sleep(1);
    }catch (InterruptedException e) {
        // fall through
    }finally {
        counterLock.unlock();
    }
}
```

- ▶ This causes the program to hang whenever it tries to decrement by an amount that is greater than count
 - ▶ Because the thread has locked counterLock no other thread can increase amount
 - ▶ This is a *deadlock*

Conditions

- ▶ Conditions allow threads to temporarily unlock locks whilst they await some condition to be fulfilled
- ▶ In this case, we'd like to temporarily unlock within a thread that is waiting to decrement
- ▶ Conditions are created from locks
- ▶ We can add a condition to MyCounter as follows:

```
private ReentrantLock counterLock = new ReentrantLock();  
private Condition bigEnough = counterLock.newCondition();
```


- ▶ Threads can await the condition through the `Condition.await()` method
- ▶ We add this to our decrement method:

```
public void decrement(int amount) {  
    counterLock.lock();  
    try {  
        while(count < amount) {  
            bigEnough.await();  
        }  
        count -= amount;  
        System.out.println("Subtracting " + amount + ", res  
    }catch (InterruptedException e) {  
        // Fall through  
    }finally {  
        counterLock.unlock();  
    }  
}
```

- ▶ A thread calling decrement when `count < amount` will wait until another thread invokes the `Condition.signalAll()` method.
- ▶ We put this method into the increment method:

```
public void increment(int amount) {  
    counterLock.lock();  
    try {  
        count += amount;  
        System.out.println("Adding " + amount + ", result " + count);  
        bigEnough.signalAll();  
    } finally {  
        counterLock.unlock();  
    }  
}
```

- ▶ Whenever an increment is made, all threads waiting on this condition are restarted.
- ▶ Note that the `signalAll()` method doesn't mean that amount is big enough
 - ▶ The syntax in `decrement()` means that `signalAll()` will cause the thread to check again.
 - ▶ It might just end up invoking `await()` again.
- ▶ Run `CounterDecounter3` and verify that count never becomes negative
- ▶ `signalAll` is analogous to `notifyAll` (`signal` is equivalent to `notify`)

wait versus await

- ▶ Using wait and notify is very similar to await and signal
- ▶ A key difference is that you could create multiple locks and conditions in a single object and have Threads waiting for different things
- ▶ Can only wait for one thing per Object With wait and notify

Threads in Swing

- ▶ In general Swing is not thread safe
 - ▶ You can't use normal threads
- ▶ But, Swing does give you threading capabilities
- ▶ First, why do we need threads in Swing?
 - ▶ `SwingThread`

- ▶ System becomes unresponsive whilst counting
- ▶ Nothing updates until counting has finished
 - ▶ until we exit the actionPerformed method
- ▶ We need threads

The event dispatch thread

- ▶ Event handling code in Swing runs on the event dispatch thread
 - ▶ e.g. `actionListeners`
- ▶ Things on this thread should be *short tasks*
 - ▶ otherwise system becomes unresponsive
- ▶ Note: some swing component methods can be invoked from any thread (marked as *thread safe* in API)
 - ▶ Why isn't all of swing thread safe? read this

Longer jobs - the `SwingWorker` class

- ▶ Long tasks should not be run on the event dispatch thread
- ▶ Instead we use worker threads
- ▶ Created by extending `SwingWorker`
- ▶ The new class must extend:
 - ▶ `doInBackground()`
- ▶ And can also use:
 - ▶ `publish()` and `process()` to display interim results
 - ▶ `done()` to invoke a method on the event dispatch thread when the task is complete.
- ▶ `Counter.java`

- ▶ Note that `SwingWorker` takes two types:
 - ▶ `SwingWorker<A,B>`
 - ▶ A: the return value
 - ▶ B: the object passed by `publish`
- ▶ Note also that `process(List b)` takes a list
 - ▶ There may be many calls to `publish` before `process` is called
- ▶ The various Swing layout things should be things you've seen before?

Other swing thread operations

- ▶ Initial threads (in SwingUtilities):
 - ▶ `invokeLater(Runnable go)` runs `go` on the event dispatch thread.
 - ▶ `invokeAndWait(Runnable go)` runs `go` on the event dispatch thread and then waits for it to finish.
 - ▶ Typically, these are used for starting the GUI (i.e. creating a `JFrame` object):

```
public static void main(String[] args) {  
    SwingUtilities.invokeLater(new Runnable() {  
        new SwingThread();  
    });  
}
```

Swing example 2 - Game Of Life

- ▶ Class exercise: building a Game of Life simulator
- ▶ Details: Conway's Game of life
- ▶ We need a responsive application that animates a 'world' and allows users to start, stop, toggle cells, change speed, clear the world and randomise the world