APIT: Concurrency

Simon Rogers

13/01/2019

Contents

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 (couting 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 sill start with general Java threads and return to Swing later

To understand why `BadFrame.java` doesn't work you have to think a bit about how things happen in Swing (something we will come back to later). But, in short, when you call something like `setText()` on a `Component` it doesn't happen immediately (althogh it previously felt like it did). In fact, it won't happen until whatever is currently happening is completed. In this example, none of the updates to the `JTextField` will happen *until* the code in `actionPerformed` has completed. So, the counting finishes and then all of the `setText()` operations are done. The same applies to detecting button clicks (etc) and updating other GUI components. For example, whilst in the `actionPerformed` method, no button clicks are processed and no other GUI updates done (this is why the button stays in its blue pressed mode until the counting has finished).

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.

For example, the following class implements `Runnable` and prints a particular `String` `n` times:

```
public class PointlessPrint implements Runnable {
    private String message;
    private int n;
    public PointlessPrint(String message,int n) {
        this.message = message;
        this.n = n;
    }
    public void run() {
        for(int i=0;i<n;i++) {
            System.out.println(i + "/" +
                n + " " + message);
        }
    }
}
```

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`

From now on, we'll use nested classes for things like this, to make the code a bit more concise:

```
public class RunnableTest {
    private static class PointlessPrint implements Runnable {
        private String message;
        private int n;
        public PointlessPrint(String message,int n) {
            this.message = message;
            this.n = n;
        }
        public void run() {
            for(int i=0;i<n;i++) {
                System.out.println(i + "/" + n + " " + message);
            }
        }
    }
    public static void main(String[] args) {
        PointlessPrint p = new PointlessPrint("Hello",100);
        Thread t = new Thread(p);
        t.start();
    }
}
```

Question: What would happen if `PointlessPrint` wasn't `static`?

Answer: You'd have to do this...

```
public class RunnableTest2 {
    private class PointlessPrint implements Runnable {
        private String message;
        private int n;
        public PointlessPrint(String message,int n) {
            this.message = message;
            this.n = n;
        }
        public void run() {
            for(int i=0;i<n;i++) {
                System.out.println(i + "/" + n + " " + message);
            }
        }
    }
    public static void main(String[] args) {
        RunnableTest2 r = new RunnableTest2();
        PointlessPrint p = r.new
            PointlessPrint("Hello",100);
        Thread t = new Thread(p);
        t.start();
```

```
    }
}
```

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();
    }
}
```

In this example, satisfy yourself that even though each new `PointlessPrint` object is stored using the same reference, they are all different.

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

---

As an aside, here are two alternative `main` methods that would do the same thing. They should both make sense to you:

```
public static void main(String[] args) {
    int nThreads = 2;
    Thread[] threads = new Thread[nThreads];
    // Declare p outside the loop if we want future access to it
```

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

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`

```java
public class SimpleThreadTest {
    private static class PointlessPrint extends Thread {
        private String message;
        private int n;
        public PointlessPrint(String message, int n) {
            this.message = message;
            this.n = n;
        }
        public void run() {
            for(int i=0;i<n;i++) {
                System.out.println(i + "/" + n + " " + message);
            }
        }
    }
    public static void main(String[] args) {
        PointlessPrint[] threads = new PointlessPrint[2];
        for(int i=0;i<2;i++) {
            threads[i] = new PointlessPrint("I am thread " + i,10);
            threads[i].start();
        }
    }
}
```

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");
    }
}
```

What's the implication?

You might think that `THE END` only appears once all the threads have finished. But actually it happends immediately after all threads have been started. This shows us that `main` is itself running on a thread and it can finish **before** the other threads. We will see how to make `main` wait using the `join()` method soon.

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()`

Example when implementing `Runnable`:

```
public class ThreadNameTest {
    private static class PointlessThread implements Runnable {
        private int n;
        public PointlessThread(int n) {
            this.n = n;
```

```
        }
        public void run() {
            for(int i=0;i<n;i++) {
                System.out.println(Thread.currentThread().getName() +" "+ i);
            }
        }
    }
    public static void main(String[] args) {
        Thread[] threads = new Thread[2];
        for(int i=0;i<2;i++) {
            threads[i] = new Thread(new PointlessThread(10),"I am " + i);
            threads[i].start();
        }
    }
}
```

Example when extending `Thread`:

```
public class ThreadNameTest2 {
    private static class PointlessThread extends Thread {
        private int n;
        public PointlessThread(int n,String name) {
            super(name); // Thread constructor
            this.n = n;
        }
        public void run() {
            for(int i=0;i<n;i++) {
                System.out.println(this.getName() +" "+ i);
            }
        }
    }
    public static void main(String[] args) {
        PointlessThread[] threads = new PointlessThread[2];
        for(int i=0;i<2;i++) {
            threads[i] = new PointlessThread(10,"Thread " + i);
            threads[i].start();
        }
    }
}
```

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()`

The following code will start a thread, wait a random amount of time and then interrupt it. When it is interrupted, it just finishes.

```
import java.util.Random;
public class InterruptTest {
    public static class InterruptableThread implements Runnable {
        public void run() {
            int i=0;
            while(Thread.currentThread().isInterrupted()==false) {
                System.out.println(i++);
            }
        }
    }
    public static void main(String[] args) {
        Thread t = new Thread(new InterruptableThread());
        t.start();
        int r = new Random().nextInt();
        for(int i=0;i<r;i++) {
            // Nothing, just eat up some time
        }
        t.interrupt();
    }
}
```

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?

After starting the first thread, the thread running `main` then waits for it to finish before starting the second one and we get no concurrency! This is a very common mistake.

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)

```java
public class CounterExample {
    public static class MyCounter {
        // We need this method because ints are immutable
        private int count = 0;
        public int getCount() {
            return count;
        }
        public void setCount(int count) {
            this.count = count;
        }
    }
    public static class Counter extends Thread {
        private MyCounter count;
        private int n;
        public Counter(MyCounter count,int n) {
            this.count = count;
            this.n = n;
        }
        public void run() {
            for(int i=0;i<n;i++) {
                int temp = count.getCount();
                temp++;
                count.setCount(temp);
            }
        }
    }
    public static void main(String[] args) {
        MyCounter count = new MyCounter();
        Counter c = new Counter(count,100);
        c.start();
        try {
            c.join();
        }catch(InterruptedException e) {
            //Do nothing
        }
        System.out.println(count.getCount());
    }
}
```

This code outputs `100` at the end, as you might expect.

Question: try removing the `join` code (i.e. everything inside the `try catch` block) - what happens?

Let's now update `main` so that it creates several threads, all accessing the same `MyCounter` object (`CounterExample2`):

```
public static void main(String[] args) {
    MyCounter count = new MyCounter();
    int nCounters = 100;
    Counter[] c = new Counter[nCounters];
    for(int i=0;i<nCounters;i++) {
        c[i] = new Counter(count,1000);
        c[i].start();
    }
    try {
        for(int i=0;i<nCounters;i++) {
            c[i].join();
        }
    }catch(InterruptedException e) {
        //Do nothing
    }
    System.out.println(count.getCount());
}
```

If all works correctly, we should see 100 times 1000 (=100000). But we don't. Each time we run, we see something different (try it), always below 100000. What's happening?

- 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...

- 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

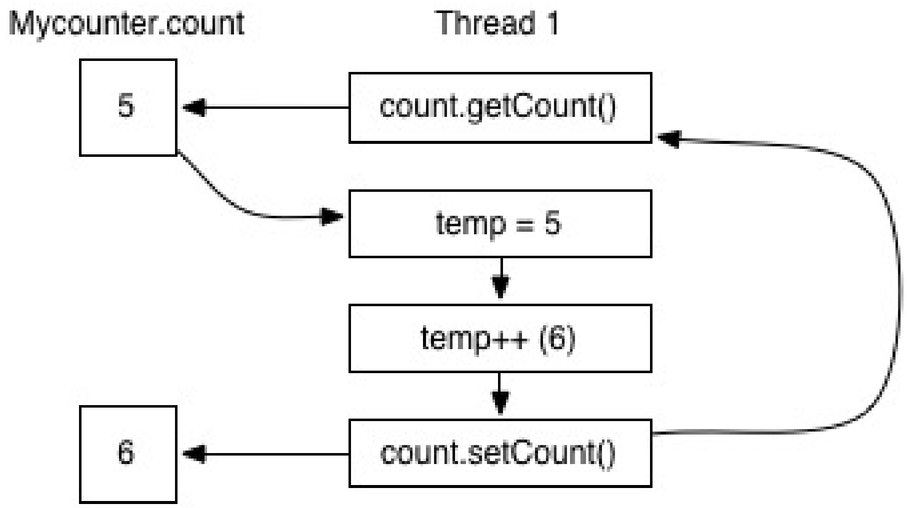

Figure 1: Single thread operation

- 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…

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.

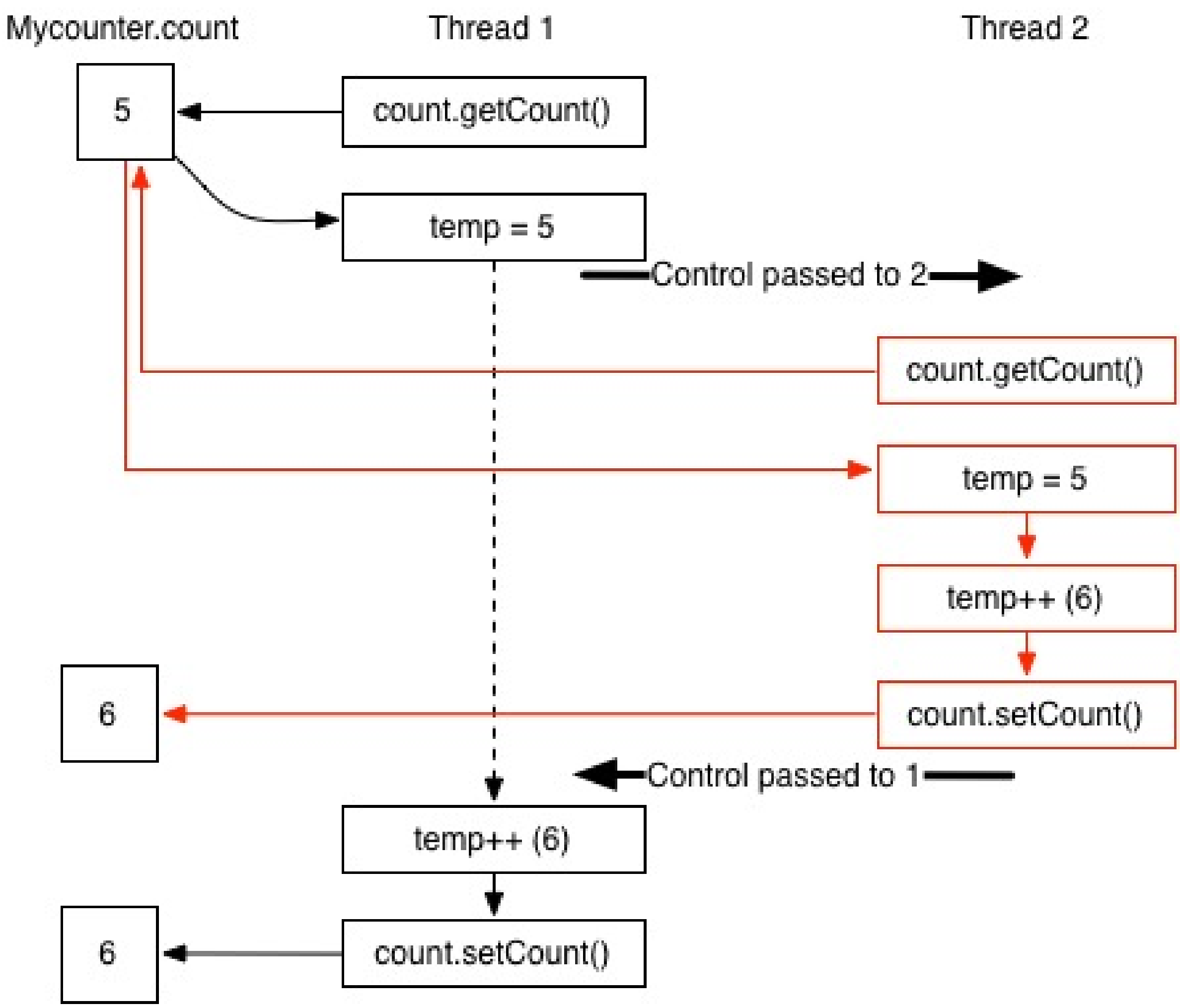

Figure 2: Multi-thread operation

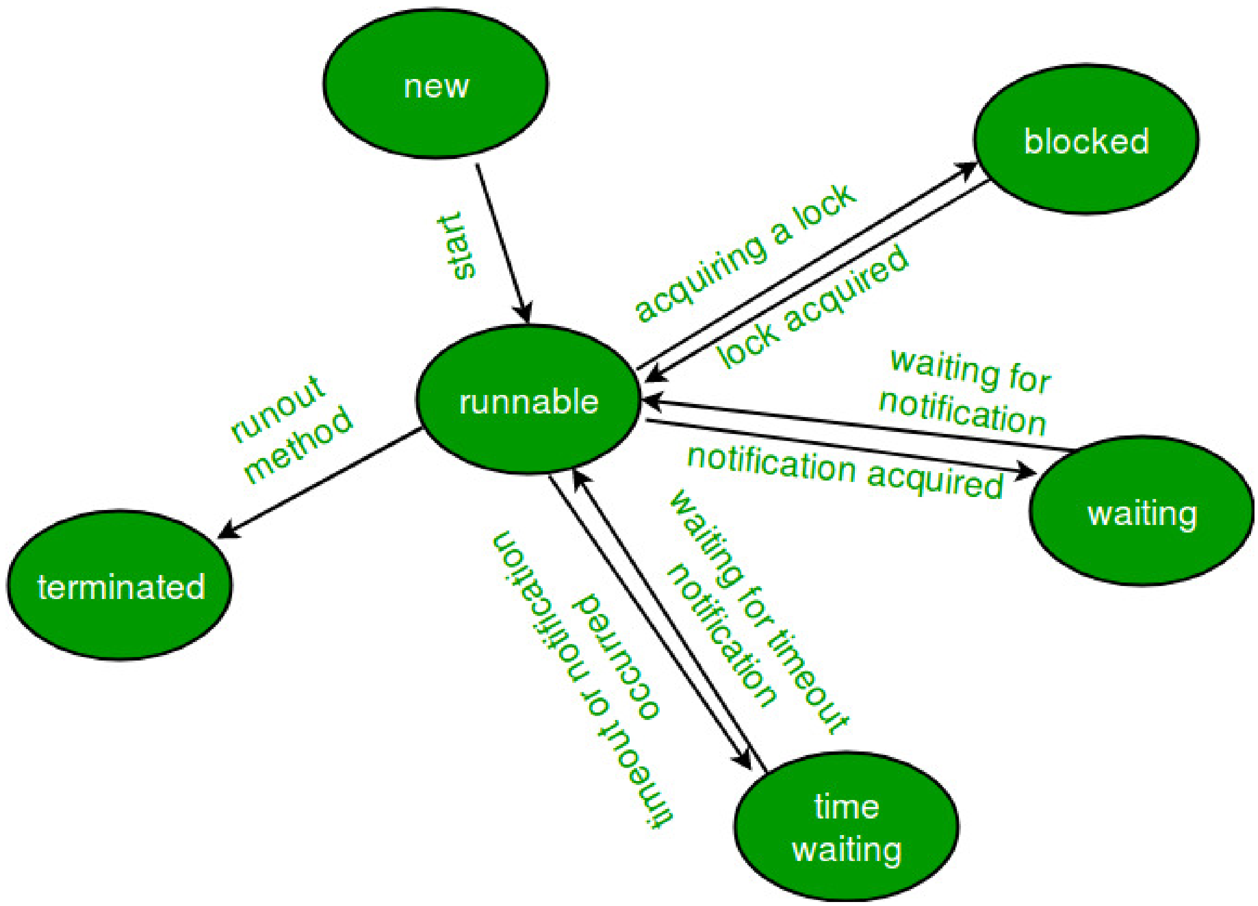

Figure 3: Thread states - Source: Core Java Vol 1, 9th Edition, Horstmann, Cay S. & Cornell, Gary_2013

- `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 problen - 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 sychronized 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 + ", result " + count);
    }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`

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JTextField;
import java.awt.GridBagLayout;

public class SwingThread extends JFrame implements ActionListener {
    private final JButton startButton,stopButton;
    private final JTextField countField,outField;
    public SwingThread() {
        super("Swing Thread");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().setLayout(new GridBagLayout());

        startButton = makeButton("Start");
        stopButton = makeButton("Stop");
```

```
        countField = makeText();
        outField = makeText();
        pack();
        setVisible(true);
    }
    private JTextField makeText() {
        JTextField b = new JTextField(30);
        getContentPane().add(b);
        return b;
    }
    private JButton makeButton(String caption) {
        JButton b = new JButton(caption);
        b.setActionCommand(caption);
        b.addActionListener(this);
        getContentPane().add(b);
        return b;
    }

    public void actionPerformed(ActionEvent e) {
        if(e.getActionCommand() == "Start") {
            outField.setText("You pressed start");
            startCounting();
        }else if(e.getActionCommand() == "Stop") {
            outField.setText("You pressed stop");
        }
    }
    private void startCounting() {
        try {
            for(int i=0;i<100;i++) {
                countField.setText(String.format("%d",i));
                Thread.sleep(100);
            }
        }catch(InterruptedException e) {

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

When this code is run, the system waits patiently until start is pressed. When this happens, control is passed to the `actionPerformed()` method. This method sets the text in `outField` and then starts the counter. During counting, the code updates `countField` and sleeps a lot.

You might guess that on pressing `start` you would see the text in `outField` immediately and then see the text in `countField` gradually increasing. Unfortunately, this is not what happens. When we call `JTextField.setText()` we are changing the text that is stored. For the screen to update, Java must do various other processes (hidden from us). These can't be done if the system is busy. In this case, the system is busy counting and so nothing changes on screen until the counting has finished, when we see the final value and 'you pressed start'.

To overcome this, we need to use the thread objects that swing provides.

- 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

In our previous code, we unwittingly put the counter on the event dispatch thread and the system became unresponsive. Only very fast things should go on this thread (e.g. starting other threads that can run in the backgroun).

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> 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?

`SwingWorker` objects allow us to run time consuming processes in the background without the system becoming unresponsive. When you extend the SwingWorker class, you *have* to implement `doInBackground()` which is the method called when we invoke the `execute()` method of the object. `doInBackground()` requires to object types to be specified: the first is its return value, the second is the type of object used by `publish()`. Other methods can also be implemented:

- `publish()`: this allows us to display preliminary results as our background task is running. Objects passed to publish can be retrieved when the system invokes `process()` ...
- `process()`: If we use `publish()` in `doInBackground()`, the system will periodically invoke `process()`. Process is passed a `List` of the objects passed to `publish()`. Often we'll only be interested in the last one. If you wanted to say update a `JTextArea` with current status, you should do so in `process()`.

- `done()`: This method is invoked on the event dispatch thread when `doInBackground()` has finished – useful for changing the enabled status of buttons etc, or displaying a message. Remember though, it's on the event dispatch thread so shouldn't be used to long tasks.

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):

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

Initial threads are typically just used for initially creating GUI objects. They put runnable objects onto the event dispatch thread. It is good practise to always create your initial objects in this manner.

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