An investigation into implicit and explicit concurrency in Java.

Dominic Rathbone

November 26, 2015

Figure 1:

```
//Runnable interface defines a thread
class AsyncTask extends Thread {
        public void run() {
                //insert code to be run in thread here
        }
}
```

Figure 2:

```
class Main {

        public static void main(String args[]) {
                AsyncTask asyncTask = new AsyncTask();
                AsyncTask.run();
        }
}
```

0.0.1 Introduction

Explicit concurrency (or explicit parallelism) is a feature of a programming language that give the programmer the ability to control the concurrency of a program by indicating which parts of it should be treated as such. In contrast, implicit concurrency is the inherent property of a programming language that allows it to handle code execution concurrently without the need of a programmer to specify how.

0.0.2 Explicit Concurrency

An example of explicit concurrency is the concept of Threads in Java. Threads are concurrently executing processes within an application, sharing the same state and memory space. By default, all applications written in java will have one main thread handling code execution. To make a program execute code concurrently, the programmer must instantiate a new instance of the Thread class which will then run the code asynchronously.

Another way of running code concurrently is by making a class implement the "Runnable" interface. Many consider this a more accurate way of representing concurrent programming in Java because all it does is specify the task that should be run by a Thread. In contrast, extending the "Thread" object is less accurate as the extension of the class implies it's behaviour is being overwriten or added upon which is not the case when you are just giving the thread a task to run.

Figure 3:

```
//Runnable interface defines a thread
class AsyncTask implements Runnable {
        public void run() {
        //insert code to be run in thread here
        }
}
```

Figure 4:

```
class Main {
        public static void main(String args[]) {
                AsyncTask asyncTask = new AsyncTask();
                Thread thread = new Thread(asyncTask);
                thread.start();
        }
}
```

Thread Pooling & Executors

As you can imagine, organising threads can become quite complicated as an application scales. To manage the creation, maintainance and destruction of threads, "thread pooling" can be used. This is a design pattern where a finite group or "pool" of threads is created, normally in the form of a queue. These threads sit idle until they are pulled from the queue to complete a task. Once this is done, they are pushed back into the queue. This has the benefit of avoiding having one thread for each task in an application which can severely hinder performance. The number of threads in this pool normally varies from application to application due to the fact it could cause bottlenecking if it is too high or too low.

In Java, thread pools can be implemented by using Executors, more specifically "ExecutorService". This is an interface implemented by several concrete classes representing several types of thread pools. The simplest of these implementations is "FixedThreadPool". Once this has been instantiated, you can submit "Runnable" objects to the ExecutorService as a task to be ran by one of the threads in the pool.

0.0.3 Implicit Concurrency

0.0.4 Issues In Concurrency

When dealing with concurrent programming, there are a number of issues.

Figure 5:

```
ExecutorService executorService =
                        Executors.newFixedThreadPool(10);

//using asyncTask from figure 1.2
executorService.execute(asyncTask);

executorService.shutdown();
```

Thread interference

One of these is thread interference when a resource is changed by one thread whilst another thread is working on it. An example of this is:

Memory Consistency Errors

Implicit Concurrency solving these

Explicit Concurrency solving these

Thread Contention

Due to the fact that resources are shared and code being executed simultaenously may try to access these resources simultaenously. Resource starvation is one of these issues where a thread is starved of a resource, constantly waiting for it to become available. A common example of this happening is when two threads become deadlocked waiting on each other to complete an action.

Implicit Concurrency solving these

Explicit Concurrency solving these