An investigation into implicit and explicit concurrency in Java.

Dominic Rathbone

December 4, 2015

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.

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();
        }
}
```

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 a thread for each task in an application which can severely hinder performance if they aren't managed correctly. 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 via an interface called "ExecutorService". This is implemented by several concrete classes representing several types of thread pools. The simplest of these implementations is "FixedThreadPool". Once this has been instantiated, "Runnable" objects can be submitted to it as a task to be ran by one of the threads in the pool.

0.0.3 Implicit Concurrency

In Java 8, the concept of implicit concurrency was introduced through the Stream API. This allows developers to work on data in parellel by converting

Figure 5:

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

//using asyncTask from figure 4
executorService.execute(asyncTask);

executorService.shutdown();
```

data structures specified in the Java Collections library to "Streams". A stream is a sequence of data that operations can be applied to as it travels through. As the streaming data is inputted into each intermediary operation, this operation outputs the data into another stream which is piped into the next and so forth until it hits the last operation which terminates, forming a "pipeline".

Figure 6: sequential streaming in Java

```
List<String> listOfStrings =
        Arrays.asList("HELLO, WORLD", "lowercase");

listOfStrings
//get stream from list
.stream()
//limit to only upper case elements
.filter(string -> string.toUpperCase().equals(string))
//print these upper case elements
.forEach(string -> System.out.println(string));
```

These intermediary operations can be executed concurrently by using a parallel stream or by calling the "parallel()" intemediary operation on a default stream. This works by forking the stream into substreams on which each operation is applied in parallel, these substreams are then joined together at the terminating operation. This

0.0.4   Issues In Concurrency

When dealing with concurrent programming, there are a number of issues to take into consideration.

Figure 7: parallel streaming in Java

```
List<String> listOfStrings =
Arrays.asList("HELLO, WORLD", "lowercase", "HELLO, DOM");

listOfStrings
//get stream from list
.stream()
//run operations in parallel
.parallel()
//limit to only upper case elements
.filter(string -> string.toUpperCase().equals(string))
//limit to strings matching phrase
.filter(string -> string.equals("HELLO, DOM"))
//print these upper case elements
.forEach(string -> System.out.println(string));
```

Thread interference

This is when a resource's value become non-deterministic due to multiple threads performing operations on it at the same time. An example of this is:

1. Counter is initialized at a value of 5.

2. Thread A increments Counter.

3. Counter is at 6.

4. Thread B decrement Counter.

5. Counter is at 5.

6. Thread A reads Counter expecting it to be 6.

7. Thread B reads Counter expecting it to be 4.

Java multithreading solves this issue with Synchronization. This is where multiple threads working on an individual resource are organized so that only one of them has access to it at a given point in time. This is handled by "monitors", each java object has a "monitor" associated with it that can be locked or unlocked. When a thread wants to access this object, it does so by locking it with this monitor, stopping any other threads from accessing it. To implement this, synchronized methods and synchronized blocks are used. The former is used to apply synchronization to the whole scope of a method whereas the latter only applies the synchronization to the block of code within it.

With parallel streaming in Java, the library controls and optimises how the stream is broken down into substreams and processed, meaning thread interference does not occur unless an operation outside the pipeline is applied to the

Figure 8: Synchronized Method

```java
public class SynchronizedStudent {

        private String name = "Dom";

        public synchronized void setName(String name) {
                this.name = name;
        }

        public synchronized String getName() {
                return name;
        }
}
```

Figure 9: Synchronized Block

```java
public class SynchronizedStudent {

        private String name = "Dom";

        public void setName(String name) {
                synchronized(this) {
                        this.name = name;
                }
        }

        public String getName() {
                synchronized(this) {
                        return this.name;
                }
        }
}
```

source collection as the pipeline is being executed, in which case a Concurrent-ModificationException is thrown.

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 deadlock,

where two threads are waiting on each other to complete an action.

Implicit Concurrency solving these

Explicit Concurrency solving these