PART A
PART 1
An investigation into implicit and explicit concurrency in Java.

Dominic Rathbone

March 19, 2016

0.0.1 Introduction

Explicit concurrency (or explicit parallelism) is a property of programming languages that give the programmer the ability to control the concurrency of their code by indicating which parts of it should be treated as such. In contrast, implicit concurrency is the property 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, away from the thread it is spawned in (see Appendix A) [1]. Another way of running code concurrently is by making a class implement the "Runnable" interface [1]. This can be considered 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 overwritten or added upon which is not the case when you are just giving the thread a task to run. (see Appendix B).

Thread Pooling & Executors

Organising threads can become quite complicated as an application scales. To manage the creation, maintenance and destruction of threads, thread-pooling can be used. This is a design pattern where a finite group 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 is decided by the developer and varies from application to application due to the fact it could cause bottlenecking if it is too high or 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 a "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.[2]. (see Appendix C).

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 parallel by converting 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 then 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. [3]. (See Appendix D). These intermediary operations can be executed concurrently by using a parallel stream or by calling the "parallel()" intermediary operation on a serial stream. This works by forking the stream into sub-streams on which each operation is applied in parallel, these sub-streams are then joined together at the terminal operation. The order that the intermediary operations are applied to each element is organised automatically by the Java runtime.[3]. (see Appendix E).

0.0.4 Issues In Concurrency

Thread interference

Thread interference is when a resource's value become non-deterministic due to multiple threads performing operations on it at the same time[4]. 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 multi-threading 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 with the notion of "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 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. [5]. (See Appendix F).

With parallel streaming in Java, the library controls and optimises how the stream is broken down into sub-streams and processed, meaning thread interference does not occur unless an operation is applied externally to the source collection as the pipe line is being applied to the stream, in which case a "ConcurrentModificationException" is thrown.[**?**]

Thread Contention

This occurs due to the fact that resources are shared between threads and more than one being executed in parallel may try to access these resources simultaneously. Resource starvation is one of these issues in which a thread is starved of a resource, constantly waiting for it to become available[6]. Another common contention is deadlock, where two threads are waiting on each other to complete an action on resource the opposite thread holds. In Java, these both can be caused by the introduction of Synchronization as it locks resources so only one thread can access it at a time.[5]

There are several ways of avoiding thread contention with explicit concurrency in Java. One of these is to use a non-lock, non-wait solution such as atomic variables. These are wrappers for primitives that implement atomic operations. These are a series of operations that are considered instantaneous by everything outside their own scope and are "all-or-nothing" in the sense that they either completely succeed or completely fail.[7] In order to avoid thread interference without synchronization, these operations implement a technique called "compare and swap" or CAS, where the memory location of the variable being changed is checked before the computation has occurred and after it has occurred. if these two values differ, it implies it has been changed by another operation and so the memory location is not updated with the new value from the computation and has to goes through the process again.[8]

Again, due to the nature of implicit parallelism, particularly with streams in Java 8, Thread Contention is not an issue as we are leaving the Java Runtime and the Streams API to deal with the organisation of multi-core processing.

Conclusion

Although the benefits of implicit parallelism over explicit in Java do seem to be numerous, it does have its limitations. In fact, benchmarks have shown it to be slower than non-parallel approaches if

used wrongly and due to it's inherent unpredictability, this is very easy to do.[3] Another problem is that all parallel streams use the same thread pool for forking and joining[10]. This means that one parallel stream could block every other one from running by submitting an expensive operation to the pool even if they are not running on the same resource. Another problem with implicit parallelism is the human aspect in that it allows developers to avoid learning how parallel processing in Java actually works by obfuscating it away and leaving the runtime and API to deal with it resulting in developers becoming more likely to use it inappropriately. It also means that if they did need to control the way a program needed to handle parallelism, it would be harder to do so.

In contrast, whilst explicit parallelism in Java has a bigger learning curve and can be inefficient if improperly utilised, it offers more freedom to experienced developers for optimisation, especially in cases where certain tasks perform better with certain concurrency strategies. On top of this, it is possible to avoid the issues of synchronization by using alternative methods to control concurrent code such as a locking framework and atomic variables.

Appendix A

Figure A.1:

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

Figure A.2:

```
class Main {

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

Appendix B

Figure B.1:

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

Figure B.2:

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

Appendix C

Figure C.1:

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

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

executorService.shutdown();
```

Appendix D

Figure D.1: 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));
```

Appendix E

Figure E.1: 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));
```

Appendix F

Figure F.1: Synchronized Method

```
public class SynchronizedStudent {

        private String name = "Dom";

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

}
```

Figure F.2: Synchronized Block

```
public class SynchronizedStudent {

        private String name = "Dom";

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

Bibliography

[1] N/A. (N/A). Threading. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency /runthread.html. Last accessed 14/01/2016.

[2] N/A. (N/A). Thread Pools. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html. Last accessed 14/01/2016.

[3] Madhusudhan Konda. (2015). Java 8 streams API and parallelism. Available: http://radar.oreilly.com/2015/02/java-8-streams-api-and-parallelism.html. Last accessed 14/01/2016.

[4] N/A. (N/A). Thread Interference. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency /interfere.html. Last accessed 14/01/2016.

[5] N/A. (N/A). Synchronization. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency /sync.html. Last accessed 14/01/2016.

[6] N/A. (N/A). Starvation and Livelock. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency /starvelive.html. Last accessed 14/01/2016.

[7] N/A. (N/A). Atomic Variables. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency /atomicvars.html. Last accessed 14/01/2016.

[8] Akhil Mittal . (2015). How CAS (Compare And Swap) in Java works. Available: https://dzone.com/articles/how-cas-compare-and-swap-java. Last accessed 14/01/2016.

[9] Angelika Langer. (2015). Java performance tutorial How fast are the Java 8 streams?. Available: https://jaxenter.com/java-performance-tutorial-how-fast-are-the-java-8-streams-118830.html. Last accessed 14/01/2016.

[10] Lukas Krecan. (2014). Think Twice Before Using Java 8 Parallel Streams. Available: https://dzone.com/articles/think-twice-using-java-8. Last accessed 14/01/2016.

PART A
PART 2
Comparing the Performance of Implicit and Explicit Concurrency in Java

Dominic Rathbone

March 19, 2016

Introduction

In order to compare the effects of concurrency on performance of a Java application, A test program was created that compared various implementations of concurrency, implicit and explicit, to a baseline non-concurrent implementation. These implementations performed a mathematical operation finding all the mersenne primes in a set of integers derived from the user inputting a start and end number at launch. Time (in nano seconds) was used as a benchmark in order to compare these implementations. The time was taken before and after each implementation ran and then the total run time was derived from them. In order to compare them with ease, a ratio derived from dividing the total time of the baseline implementation by the total time of concurrent implementation is outputted to csv files. To run the programs, A ThinkPad laptop running Fedora with a quad-core 2.50GHz i5-2520M CPU and 8 gigabytes of RAM was used as well as a Gaming PC running Windows 7 with a quad-core 3.50GHz i5 4690K and 8 gigabytes of RAM.

0.1 Non-concurrent Approach

The non-concurrent approach runs the operations sequentially, running over the sequence of integers with a for statement and incrementing a counter every time a prime number is found. This is used as the baseline to compare the implicit and explicit concurrent approaches to.(see Appendix A)

0.2 Multi-threading

The first explicitly concurrent implementation uses raw threads to run the operation over the sequence of numbers. This was attempted in two ways. First of all, to make the test a better comparison, I produced an implementation that spawned a new thread for every number in the sequence (see Appendix B, figure 1), similar to how a parallel stream would split a stream of numbers into multiple parallel sub-streams. On testing this implementation, it was revealed that this would not be an efficient way to approach concurrency because as the sequence of numbers became larger, it would become extremely slow (finishing minutes where other approaches finished in seconds). This was due to the lack of thread management such as a queue or limit to how many threads should be started at once meaning the machine bottlenecked by a flood of threads. As a result of this, the implementation was re-factored and it was decided that the best way to approach this was to split the sequence of numbers down into smaller sequences using the number of cores with one subsequence per thread per core (see Appendix B, figure 2). This would mean that although each CPU core had to process with a larger sequence of numbers per thread, the number of threads and the overhead associated with them was lower making the time drastically faster.

0.3 Thread Pooling

The second explicitly concurrent implementation was through the use of thread pools and futures. A thread pool is a group of pre-defined threads that can be used to run tasks. In this approach, a task is submitted to the thread pool for every number in the sequence. This is similar to the raw multi-threaded implementation but the thread pool can queue the threads so they do not start at the same time and bottleneck the machine as mentioned in the previous section. The results of this task is then retrieved through a "Future", an object that represents the results from the submission of the task (see Appendix C).

0.4 Streaming

This approach takes the sequence of number and processes it as a stream, applying intermediate operations on each element in the stream until it reaches an terminal operation which ends the set of operations. This can be done sequentially with a serial stream (see Appendix D) or concurrently with a parallel stream (see Appendix E). The result set includes both serial and parallel stream in order to compare how they perform.

0.5 Data Analysis

From the data (Appendix F), we can see that running the program on the desktop PC gave better results on average compared with running them on the laptop with the highest ratio for each concurrent implementation running on PC being 1.5x to 2x faster than on the laptop and the average ratio being 1.5x faster on the PC than on the laptop. This implies that all of these concurrent implementations do scale with hardware and that, vice versa, worse hardware will bottleneck concurrent applications. However, there are other variables that could have affected the performance such as other processes using the CPU during the period the program was running or the different operating system that the tests were being ran on (Windows 7 & Fedora 22).

Across the implementations, on both PC and laptop, it can be seen that on average, The raw multi-threaded approach and the parallel stream were faster compared to the baseline than the thread pool approach especially as the size of data set got larger. However, the graphs (Appendix G) indicate that on both machines, as the end value got larger, the thread pool can be seen to improve dramatically in performance, in particular when the end value reached 801million. This could imply that the programs were not ran over a data set with a large enough end value and that the thread pool may actually perform just as well or better as the numbers increased past this point. This is especially worth noting as the highest ratio (1:3.7972339477 on PC, 1:2.0458180569 on laptop) from the thread pool data found at the 800-801 million data set is higher than the highest ratio from the parallel stream approach (1:3.5239193257 on PC, 1:2.0271960552 on laptop). On the other hand, if we look at the graph mapping the size of the data set to performance, the thread pool approach seems to peak at a size of 1,000,000 to 3,000,000 and then drop as the size increases past this point implying that although it may perform at larger numbers, performance might decrease as the distance between the start value and the end value gets larger.

It can be seen from the graphs (Appendix G) that, across all concurrent implementations and machines, as the end value of the data sets increased, the performance initially increased until a plateau was reached at around the size of 200000. The increase could be because as the numbers got larger, the cost of the overhead of running the tasks concurrently in each implementation was outweighed by the benefits of calculating these tasks in this manner. The plateau implies that the software reached a point where hardware limitations such as the number of cores started to bottleneck the program.

One unusual result was how the performance of the multi-threaded approach spiked twice very early on in the graph which is surprising as the cost of using threads on data sets this small usually outweighs the benefit. This occurred on both the desktop PC and the laptop results which implies that it isn't caused by something machine-specific.

Overall, the results reflected what was expected from the tests with the exception of the thread pool approach, of which I expected to be faster than the multi-threaded approach due it's thread management capabilities. However, due to how I re-factored the multi-threaded approach to split the sequence of numbers differently, it may not be fair to compare the two. If I were to leave the multi-threaded implementation to run the operations in a similar manner to the thread-pool, it would most likely vastly be out-performed.

Appendix A

Figure A.1: Non-concurrent

```
public static void nonConcurrent() {
        for ( long n = START; n < END; n++ ) {
                int primeCount = 0;
                if (Calculator.isMersennePrime(n)) {
                        primeCount++;
                }
        }
}
```

Appendix B

Figure B.1: Multi-threaded 1

```
public static void multiThreaded() {
        List<Thread> threads = new ArrayList<>();
                for (long n = START; n < END; n++) {
                        Thread thread = new Thread(
                                new RunnableA(n)
                        );
                        thread.start();
                        threads.add(thread);
                }
                for(Thread thread: threads) {
                        try {
                                thread.join();
                        } catch (InterruptedException e){
                                e.printStackTrace();
                        }
                }
}
```

Figure B.2: Multi-threaded 2

```
public static void multiThreaded2() {

        List<Thread> threads = new ArrayList<>();
        long subsequence = (END - START) / CORES;

        for ( long n = START; n < END; n=n+subsequence) {
                Thread thread = new Thread(
                        new RunnableB(n, n+subsequence)
                );
                thread.start();
                threads.add(thread);
        }

        for(Thread thread: threads) {
                try {
                        thread.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Appendix C

Figure C.1: Thread Pool

```
public static void threadPool() {
        ExecutorService executorService =
                Executors.newFixedThreadPool(CORES);
        ArrayList<Future<Integer>> futures =
                new ArrayList<>();

        for ( long n = START; n < END; n++ ) {
                final long number = n;
                Future<Integer> future =
                executorService.submit(() -> {
                  if(Calculator.isMersennePrime(number)) {
                          return 1;
                  }
                  else {
                          return 0;
                  }
                }
                );
                futures.add(future);
        }

        int primeCount = 0;
        for (Future<Integer> future : futures) {
                try {
                        primeCount += future.get();
                } catch (Exception e) {
                }
        }
        executorService.shutdown();
}
```

Appendix D

Figure D.1: Serial Stream

```
public static void serialStream() {
        long primeCount = Stream
        .iterate(START, n -> n + 1)
        .limit(END-START)
        .filter( n -> Calculator.isMersennePrime(n) )
        .count();
}
```

Appendix E

Figure E.1: Parallel Stream

```
public static void parallelStream() {
        long primeCount = Stream
        .iterate(START, n -> n + 1)
        .limit(END-START)
        .parallel()
        .filter(n -> Calculator.isMersennePrime(n) )
        .count();
}
```

Appendix F

results_4_CORES_PC_BY_END_VALUE

START VALUE	END VALUE	NC/MT ratio	NC/TP ratio	NC/SS ratio	NC/PS ratio	RANGE
1000	2000	3.377639498	0.31913372	0.027929964	0.034146773	1000
1000	2000	0.320807279	0.055333735	0.286081339	0.252846387	1000
1000	2000	0.239644043	0.068624413	0.357876721	0.172952854	1000
1000	2000	0.196722643	0.067835434	0.394739209	0.194176127	1000
1000	2000	0.225326922	0.073045897	0.398684603	0.200000565	1000
1000	2000	1.758723902	0.253959165	0.024393171	0.023739355	1000
1000	2000	0.217272688	0.041493072	0.250262166	0.165512344	1000
1000	2000	0.220710359	0.051581958	0.374426602	0.19209725	1000
1000	2000	0.170696852	0.060717377	0.328892813	0.163466585	1000
1000	2000	0.210390246	0.074173599	0.425127366	0.220730944	1000
10000	20000	0.858815407	0.158652262	0.725508847	0.51270343	10000
10000	20000	1.205827301	0.173652192	0.823844341	0.465417258	10000
10000	20000	1.176961127	0.262230756	0.816759772	0.747987943	10000
10000	20000	1.151818063	0.196830896	0.825228021	1.050290563	10000
10000	20000	1.188513651	0.269173537	0.859399262	0.878229009	10000
10000	20000	0.766828061	0.13161994	0.667895594	0.566473385	10000
10000	20000	1.181371339	0.253831024	0.805998556	0.484615564	10000
10000	20000	1.859411256	0.245705539	0.831074158	0.799089241	10000
10000	20000	1.207935607	0.188525057	0.82735425	1.078820341	10000
10000	20000	1.19423832	0.284880282	0.852876299	0.936588513	10000
100000	200000	2.340828048	0.506817026	0.9082702	1.865600518	100000
100000	200000	2.125376932	0.416007445	0.963233115	2.219974061	100000
100000	200000	3.293454721	0.595493681	0.957305169	2.259292598	100000
100000	200000	3.368788592	0.615754195	0.957900049	2.594866714	100000
100000	200000	3.231993278	0.624263085	0.966778684	2.051144435	100000
100000	200000	2.258560136	0.513285156	0.926996047	2.006472467	100000
100000	200000	3.415948165	0.434204296	0.977496539	1.64506166	100000
100000	200000	3.139018057	0.612744375	0.958686746	2.246692809	100000
100000	200000	3.51202543	0.644250985	0.991885105	2.71182824	100000
100000	200000	3.214072594	0.62013467	0.951310813	2.217843599	100000
1000000	1300000	4.451915331	1.278310212	1.006379802	1.322046076	300000
1000000	1300000	3.856927662	1.260232203	0.970579024	3.523919326	300000
1000000	1300000	3.792950079	1.159014021	0.950371928	3.258594083	300000
1000000	1300000	3.770619274	1.223875962	0.9181477	3.355876674	300000
1000000	1300000	3.794271987	1.382822903	0.959746074	3.448348137	300000
1000000	2000000	3.514506636	1.421082941	0.985529534	3.073749118	1000000
1000000	2000000	3.519729814	1.671682961	0.967841018	3.205534829	1000000
1000000	2000000	3.624065796	0.745356153	0.978514607	3.192181753	1000000
1000000	2000000	3.614826372	1.572081096	0.955582479	3.203334201	1000000
1000000	2000000	3.619815149	1.523739953	0.976732052	3.148937508	1000000
1000000	2000000	3.517761987	1.312032542	0.982016502	3.079629016	1000000
1000000	2000000	3.588292519	1.587881331	0.988058455	3.078592823	1000000
1000000	2000000	3.616462549	1.408812218	0.980326377	3.17659234	1000000
1000000	2000000	3.582887706	0.749203995	0.97802057	3.170586645	1000000
1000000	2000000	3.633200606	1.664774504	0.983058502	3.23337179	1000000
4000000	16000000	3.362055883	1.046243234	0.993858044	2.034823663	12000000
4000000	16000000	3.382761937	1.334551493	0.9883842	3.118080474	12000000
4000000	16000000	3.412241354	1.735529724	0.958752682	3.065004549	12000000
4000000	16000000	3.243706726	1.871174476	0.952584622	3.028928073	12000000
4000000	16000000	3.469829834	2.470716633	0.946710912	3.069480705	12000000
10000000	20000000	3.748113149	1.747108969	0.999592546	3.266074489	10000000
10000000	20000000	3.695950777	2.177803496	0.995090801	2.470590206	10000000

10000000	20000000	3.692996818	1.917985992	0.98914144	2.476299678	10000000
10000000	20000000	3.685017147	1.634481984	0.999603464	3.160641051	10000000
10000000	20000000	3.141016189	1.471266483	1.006922844	3.161235007	10000000
10000000	20000000	3.723301464	1.86687083	0.989056198	3.172549963	10000000
10000000	20000000	3.716738113	2.277855382	0.994316709	3.129259425	10000000
10000000	20000000	3.645727715	2.403086301	0.994768397	2.911261845	10000000
10000000	20000000	3.365285863	2.375225615	0.991145406	3.062920633	10000000
10000000	20000000	3.725905115	2.305160755	0.986125472	3.025711226	10000000
800000000	801000000	3.362653602	3.267070376	0.991573219	3.246680993	1000000
800000000	801000000	3.623860248	3.452837263	0.993757194	3.257731374	1000000
800000000	801000000	3.643522237	3.643523353	1.024524142	3.297184018	1000000
800000000	801000000	3.687123738	3.750505077	1.027722113	3.277188692	1000000
800000000	801000000	3.716744776	3.797233948	1.030373262	3.354370366	1000000

results_4_CORES_LAPTOP_BY_END_VALUE

START VALUE	END VALUE	NC/MT ratio	NC/TP ratio	NC/SS ratio	NC/PS ratio	RANGE
1000	2000	1.822415532	0.174783803	0.021748569	0.1860857	1000
1000	2000	0.161616839	0.024289526	0.253046546	0.119190818	1000
1000	2000	0.048011791	0.04524044	0.325850436	0.090595202	1000
1000	2000	0.403661601	0.065935929	0.582434319	0.154057565	1000
1000	2000	0.284183221	0.06703164	0.552081013	0.159012571	1000
1000	2000	1.842330849	0.160027263	0.019974901	0.159624162	1000
1000	2000	0.177369389	0.031642014	0.310330343	0.098413259	1000
1000	2000	0.012354499	0.0307123	0.276183797	0.100820143	1000
1000	2000	0.077390854	0.066994761	0.391656298	0.164510317	1000
1000	2000	0.203584745	0.03672512	0.49585795	0.1425688	1000
10000	20000	0.234928128	0.100118398	0.729529159	0.448235394	10000
10000	20000	0.570385742	0.151337512	0.984669945	0.634057303	10000
10000	20000	0.319487332	0.24590822	0.625115128	0.471049947	10000
10000	20000	1.137688373	0.189088368	0.80783934	0.746968622	10000
10000	20000	0.950581139	0.177498382	0.822928318	0.843591044	10000
10000	20000	0.509015271	0.098484672	0.450664007	0.313595908	10000
10000	20000	0.978315001	0.185763271	0.553286727	0.38131843	10000
10000	20000	0.278352671	0.259175352	0.579025524	0.387977223	10000
10000	20000	1.158372939	0.159446671	0.830768952	0.881910015	10000
10000	20000	0.9439115	0.143231888	0.855080441	0.874236673	10000
100000	200000	1.771160604	0.385661468	0.946951505	1.001947547	100000
100000	200000	1.479750051	0.428212203	0.90944275	0.898461912	100000
100000	200000	1.283563869	0.404342675	0.975244854	1.580723771	100000
100000	200000	1.705189941	0.389480979	0.972299909	1.64552861	100000
100000	200000	1.976323106	0.459602293	0.959332167	1.620017516	100000
100000	200000	1.595393784	0.513201657	0.873398782	1.069756077	100000
100000	200000	1.664986356	0.345129971	0.87739217	0.928963124	100000
100000	200000	1.467217539	0.50376251	1.003307829	1.646249868	100000
100000	200000	1.880497996	0.38833524	0.964506042	1.565578952	100000
100000	200000	1.995171915	0.45527218	0.975070727	1.529786788	100000
1000000	2000000	2.089357622	0.791744079	0.963785758	1.796101601	1000000
1000000	2000000	1.988806744	0.482402106	0.883167699	1.755362029	1000000
1000000	2000000	2.050816105	0.741354332	0.981150038	1.785105642	1000000
1000000	2000000	2.081661363	0.960609993	0.959895347	1.795616736	1000000
1000000	2000000	1.898397087	0.901628952	0.969452616	1.727401005	1000000
1000000	2000000	2.09032314	0.851776595	0.971019988	1.865149783	1000000
1000000	2000000	2.046910691	0.463840128	0.956443561	1.868196816	1000000
1000000	2000000	2.077055987	0.720316127	0.967484725	1.696934362	1000000
1000000	2000000	2.040108794	0.841348667	0.968571073	1.904896911	1000000
1000000	2000000	2.074644999	0.788702468	0.947120161	1.732486578	1000000
10000000	13000000	2.126506745	0.952776447	0.98156175	1.295640381	3000000
10000000	13000000	1.983727574	0.936264658	0.951409062	1.878472587	3000000
10000000	13000000	2.060761698	1.384711071	0.979501647	1.833994618	3000000
10000000	13000000	1.971154365	1.37110004	0.954518667	1.836056547	3000000
10000000	13000000	2.120054759	1.320480343	0.984174719	1.857594647	3000000
4000000	16000000	2.069670227	0.736040106	0.970988022	1.22231773	12000000
4000000	16000000	1.866702102	0.782743563	0.93249456	1.728418534	12000000
4000000	16000000	1.914680848	0.755840154	0.960232511	1.515119466	12000000
4000000	16000000	2.015967698	1.080844308	1.0379565	2.027196055	12000000
4000000	16000000	1.93975733	0.880806754	0.967207957	1.787200522	12000000
10000000	20000000	1.984645582	0.896206526	0.977812771	1.645463362	10000000
10000000	20000000	1.904503518	0.886975958	0.959679234	1.844449905	10000000

10000000	20000000	2.032640562	1.05033448	0.975789427	1.878804544	10000000
10000000	20000000	1.977990641	0.946475655	0.975593993	1.787809845	10000000
10000000	20000000	2.106130586	1.127567337	0.970219037	1.861740716	10000000
10000000	20000000	2.047182357	0.893938868	0.983768078	1.587451144	10000000
10000000	20000000	1.996461115	0.981548645	0.966075405	1.828673122	10000000
10000000	20000000	2.094017136	1.131197097	0.968992706	1.853934805	10000000
10000000	20000000	2.030319183	1.145986953	0.978175409	1.800510511	10000000
10000000	20000000	2.091426473	1.086843941	0.97335341	1.567636952	10000000
800000000	801000000	2.158586995	1.900791289	1.010628296	1.986966865	1000000
800000000	801000000	2.026315654	1.935729615	0.933755046	1.833623535	1000000
800000000	801000000	2.100828807	2.007524164	0.985084759	1.967674223	1000000
800000000	801000000	2.110788815	2.045818057	0.989801464	1.942106473	1000000
800000000	801000000	2.107337597	1.993997269	0.984795236	1.941794331	1000000

Appendix G

Figure G.1: PC Results

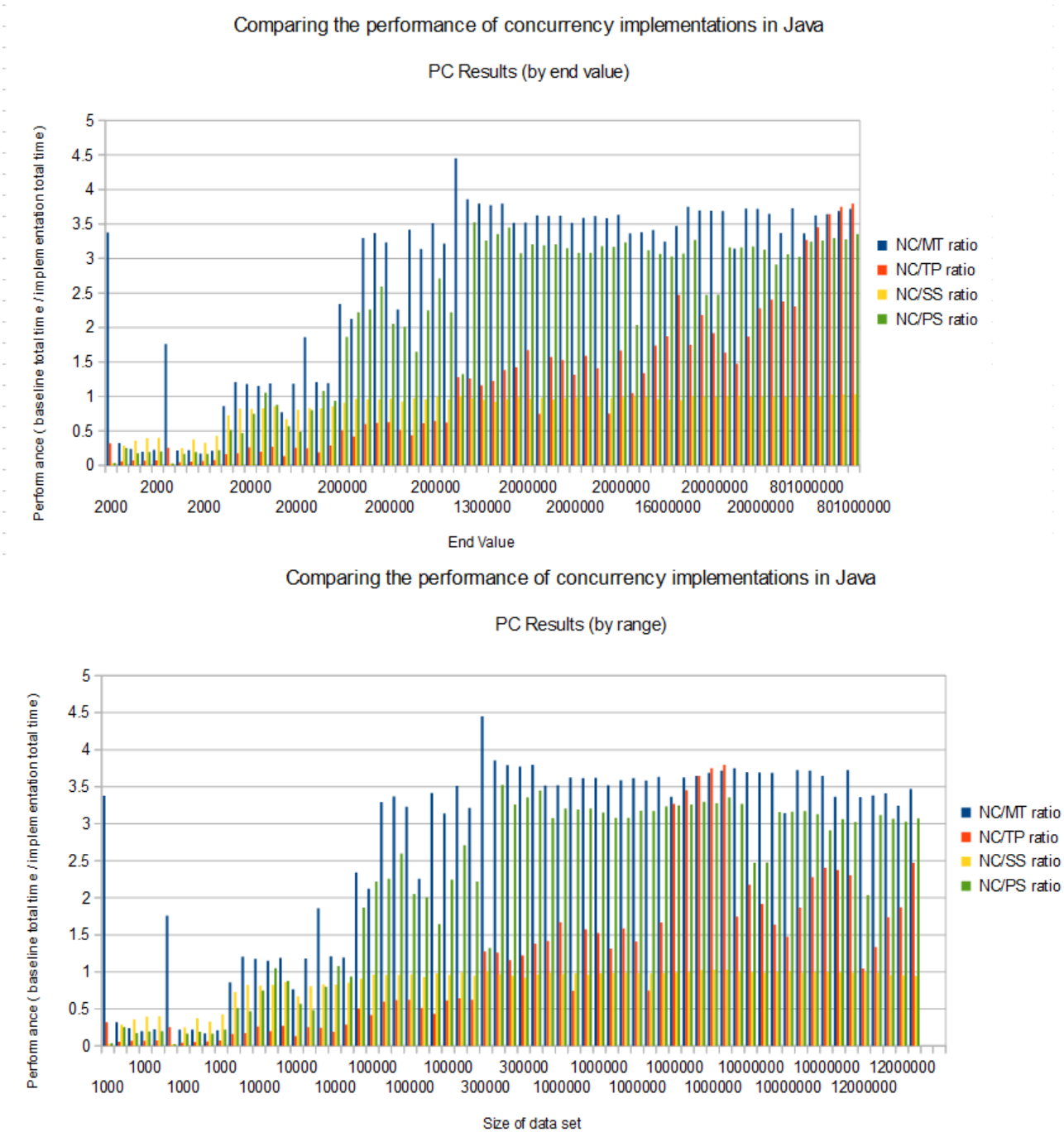
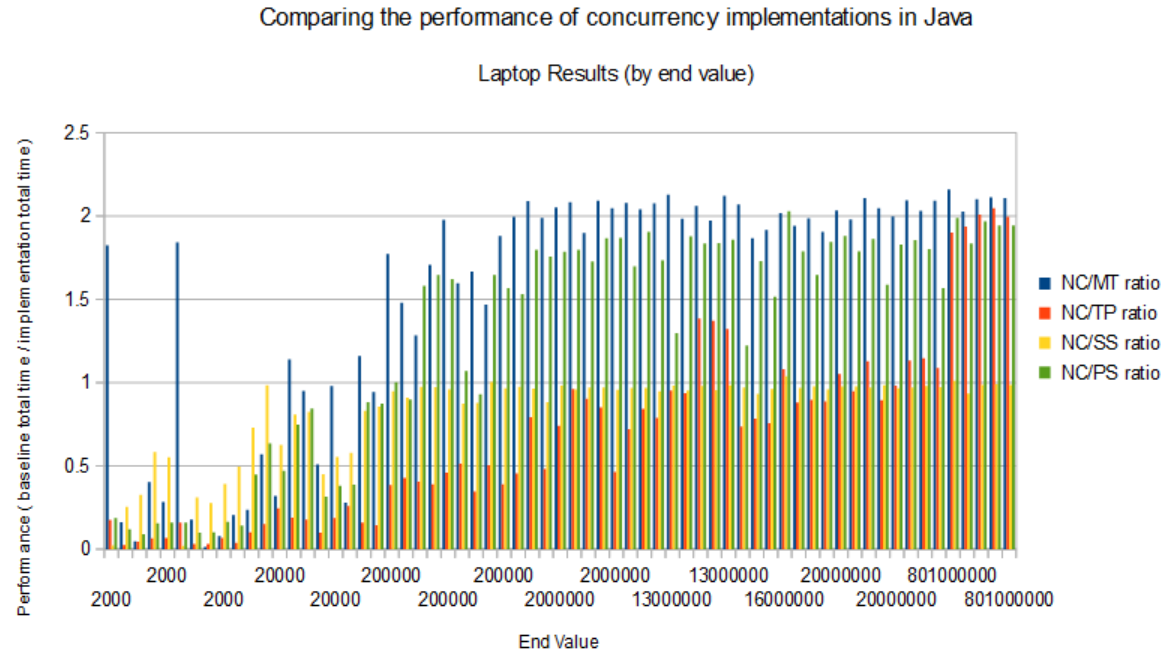
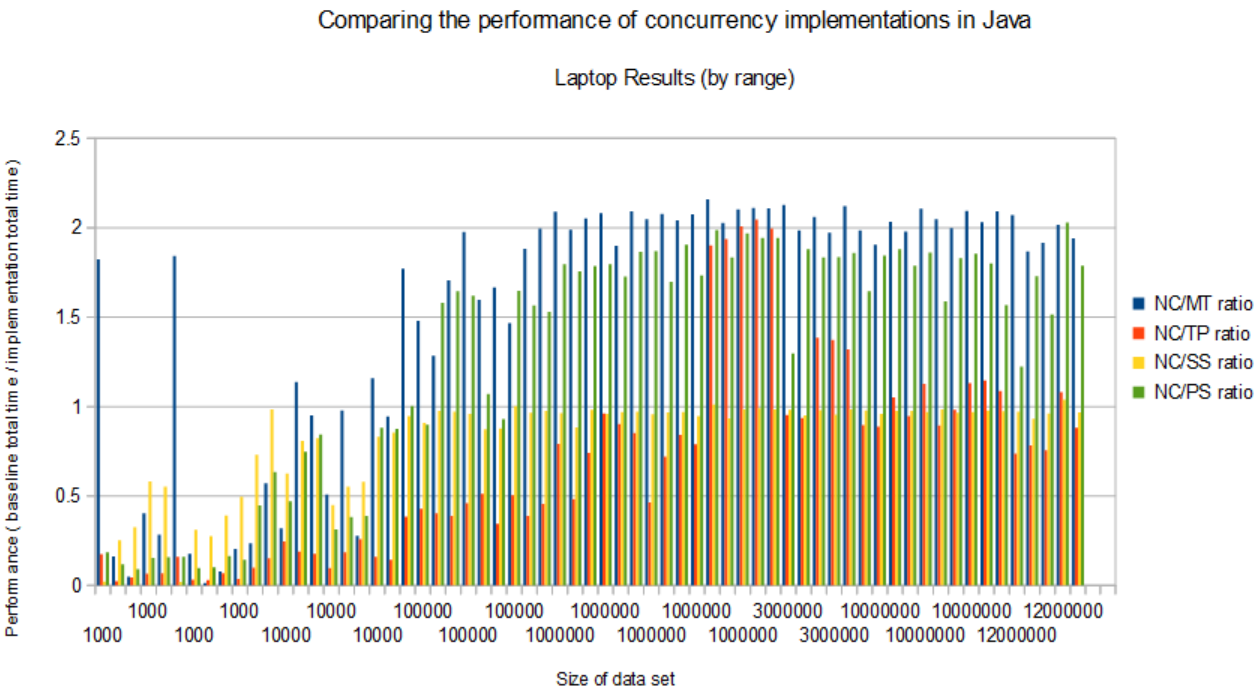


Figure G.2: Laptop Results





PART B
A Comparison of C++ and Java.

Dominic Rathbone

March 19, 2016

0.1 Introduction

The aim of this report is to compare and contrast the programming languages, C++ and Java. This comparison will be based around a number of aspects that define a programming language such as it's type system, how it is compiled and the paradigm's it utilises.

0.2 History

C++ was developed by a Computer Scientist named Bjarne Stroustrup, arising from an early project of his, "C with classes". The intention of this was to produce a superset of the C language that supported the object oriented . His inspiration for this came from his experience with the programming language, "Simula 67" which exposed him to the object-oriented programming paradigm. Eventually, in 1983, C With Classes evolved into C++ with a new feature set being added. [1]

On the other hand, Java was produced at Sun Microsystems (Now, Oracle) in secret by a team known as the "Green Team" with the intentions of creating a modern alternative to C++, the most popular language at the time. The idea was spawned out of the green team's frustrations with the C/C++ APIs that Sun had been using and aimed to retain most of the feature set that C++ had whilst removing the unnecessary, outdated parts. During the process of becoming Java, it went through two iterations of names, "C++ ++–" and then Oak.[2]

0.3 Paradigms

As mentioned above, both C++ and Java are languages that support the object oriented paradigm. This concept describes how a system can be modelled as structures known as "objects" that contain the data that describe them as well as the operations that manipulate and use this data. This is based on the imperative programming paradigm, where a language has the ability to control the flow and state of a system. [3]

0.4 Typing

Java is considered statically typed as the code is checked for type errors at compilation time. This is in contrast to dynamic typing where type-checking happens at runtime where errors will only occur if the code containing them is executed. It can also be considered strongly typed (relative to other languages) as it has a safe type-system where a variable declared as one type cannot be initialised or changed to be another type. For example, if a new variable is initialised as an integer and then a string is added to it (see Appendix A), the compiler will throw an error as the types are mismatched. Whilst static and strong typing restricts what a programmer can do with the code, it has the obvious benefit of preventing potentially serious errors from occurring.[4]

Although C++ is statically typed, it can be said to be weakly typed as types can be be implicitly converted from one to another through a variety of means. For example, a short can be implicitly cast to an integer because they are both of numerical types (see Appendix B). It also allows implicit casting of any type pointers to a void pointer (one which contains no type information). However, in contrast to it's predecessor C, it doesn't allow conversion from void pointers to a type pointer without an explicit cast (see Appendix B). This is an example of how it could be argued that C++ is strongly typed (in comparison to C). [4]

Both languages contain the notion of primitive types which are basic, predefined types such as integers or booleans which have the purpose of acting as building blocks within the language.

0.5 Arguments & Parameters

In Java, objects passed into a method via it's parameters are pass-by-value as opposed to pass-by-reference. Pass-by-value means that when you pass through an argument to a function, it passes through a copy of the pointer to that argument. In Java, this means that when you pass through an object into a method, the value of the parameter inside the method will initially reference that object but it is not the object itself. (see Appendix C) [5].

In C++, you can pass parameters either via pass-by-value (see Appendix D, figure 1) or via pass-by-reference. Pass-by-reference means that when you pass an argument into a function, the parameter inside will reference the actual argument being passed through. This means changes made inside the function scope will effect the object outside the function scope. To pass by reference in C++, you use the address-of operator, symbolized by an ampersand (see Appendix D, figure 2) [6].

0.6 Memory Management

Memory can be split up into parts, predominantly the stack and the heap. The former is the memory which stores predetermined values such as function calls and local variables that are not used outside of a functions's scope. The latter is considered dynamic memory, where dynamically created variables such as objects are stored.

Java can be considered memory-safe as you can not directly control how this memory is managed. Instead, the Java virtual machine has garbage collection. This is the process of automatically freeing up dynamic memory for re-use at runtime by tracking live objects whilst declaring everything else as garbage taking up memory. The JVM does this by allocating a new object space in the heap defined at it's start up. When the object goes unreferenced, the garbage collector reclaims the memory in this space and it is available for reuse by other objects. Although this is efficient, the space reclaimed by the garbage collector isn't ever actually given back to the operating system and the Java program will always take up the amount of initial heap space defined at the start which can cause potential performance problems if this space is too large, especially when considering the process already takes up more resources as it is automated [7].

On the other hand, in C++, memory isn't handled by a virtual machine. The task of allocating the addresses at which a variable's value should be held is handled by the operating system. In C++, managing the object's creation and deletion to and from dynamic memory is a manual task which is achieved by the new and delete operators (see Appendix E). However, in a big system, this can be a laborious task to do manually and if done incorrectly, it can result in memory/resource leaks. To counter this, there are techniques such as smart pointers that wrap raw pointers and handle the creation and deletion automatically [8]. Another technique is explicit garbage collection, although C++ lacks this implicitly, it can be efficiently implemented via libraries such as the "Boehm collector" library. [9].

0.7 Compilation

With Java, compilation is handled by a compiler called javac. This converts the raw Java source code to byte code, a series of instructions contained within a class file that are then interpreted by the JVM and compiled to native machine code at runtime (this is called Just-In-Time compilation). However, dependent on implementation of the JVM, Java code can be interpreted or compiled in many ways. The key advantage and purpose of the JVM is that it means Java can be ran on any machine as it doesn't use a platform-dependent compiler. [10]

C++ differs in this sense as it uses platform-dependent compilers so if code is written that uses a library specific to mac, it would compile for mac it most likely wouldn't on a windows machine. Although it lacks cross platform functionality, this means the machines don't have the overhead of

running the JVM, making it more performant. It also gives more freedom to the developer as they are closer to the machine and thus can optimise resource/memory usage more precisely. The process of compilation for C++ can be broken down into 4 steps, preprocessing, compilation, assembly and linking. Preprocessing is the step where directives such as "#include" are extracted from each C++ file and handled, leaving behind plain C++. The compilation process takes this output and parses it into an assembler file which the assembler takes and creates an object file with. The linking step takes these object files and links them together to produce an executable file. [10]

0.8 Concurrency

Concurrency can be explained via the concept of multi-threading. A Thread represent a series of tasks that need to be executed and thus multiple threads represents multiple series of tasks that can be executed. Of course, this can cause problems as threads share resources such as data which becomes non-deterministic in a multi-threaded environment as it can never be guaranteed to be the same value for every thread, these are called race conditions.

By default, a program written in Java will use a single "main" thread and all tasks will be executed from this thread. However, a developer can create and delete new threads from this "main" thread in order to run multiple series of tasks in parallels (see Appendix F, Figure 1). This is where the race condition used as an example earlier occurs. This can be solved in Java with thread synchronization. Synchronization uses monitors associated with objects to lock and unlock access to parts of that object. Only one thread can hold this monitor at a time so only it can change the data inside this part. Synchronization can take form of a synchronized method (See Appendix F, Figure 2) or a synchronized code block (See Appendix F, Figure 3).[11]

C++ is similar to Java in how threads are created and deleted. To solve the race condition that comes from with resource sharing in a multi threaded environment, C++ uses a "mutex" (mutual exclusion) object containing a lock and unlock function. The lock function is called at the beginning of the block of code that only should be ran in one thread and the unlock function is called at the end, preventing multiple threads from accessing it at the same time (see Appendix G).[12]

In both languages, it is also possible to use promises and futures with asynchronous tasks to avoid having to explicitly create threads. In Java, this is achieved by creating an executor service and then submitting the task that needs to be ran asynchronously to the executor. The result of the submit method is called a "future" which you then call a method from to retrieve the result (see Appendix H). In C++, this is achieved by using the "async" construct, the reference value of the function that is to be ran is passed through to it along with with the arguments that are to be used with the function. The result of "async" is a future which can be used to retrieve the result from at any time (see Appendix I).

In Java, it is possible to set the priority the threads have. This indicates how much computation time the threads get from the CPU. In Java, this is simply done by calling the "setPriority" method on the Thread object and the higher the number, the higher the priority (see Appendix F, figure 1). In C++, it is not possible to set thread priority with the standard library as this is handled by the operating system.

0.9 Error Handling

Error handling in Java is handled with exceptions (short for "exception events"). These are events that disrupt how the program would normally work. There are two types of exceptions, checked and unchecked exceptions. Checked exceptions are exceptions that have to be caught by a method somewhere and are detected at compilation time. An example of this is a FileNotFoundException. This is detected at compile time if the file is not there and has to be dealt with before the program will compile. Unchecked exceptions are exceptions that are undetected at compile time and occur

at runtime. An example of this is ArrayIndexOutOfBoundsException which is unchecked because it cannot be detected at compile time if the array size is constantly changing. It is optional to catch these exceptions but if they are not caught anywhere, they will bubble up the runtime stack and eventually cause the program to stop executing. To catch these exceptions, you can either throw the exception and rely on the code catching it somewhere else in the stack or you can use a try/catch block to wrap the block of code that might cause an exception (see Appendix H for an example).[13]

Modern C++ has very similar error handling to Java, using exceptions to handle errors. One difference is that it doesn't have the notion of a checked exception. All exceptions are unchecked and are optionally caught (see Appendix L). This can be considered detrimental as exceptions only occur at runtime and so may never occur until it is too late. The syntax for throwing, trying and catching these exceptions is similar to Java.[14]

0.10 Reflection

It is hard to compare programming languages because they all fit their own niches and purposes. Both Java and C++ can be considered general purpose languages but they are definitely optimised for different purposes.

C++ is closer to the system due to the lack of a VM which gives it less of an overhead to run and thus can be more performant with, for example, embedded systems which have limited resources to run on. It also allows developers more freedom for optimising how their code works with a system as they use things such as OS-specific libraries. However, in the modern age with CPUs getting smaller and faster, it is useful to note that this advantage might not be worth the costs associated with it such as the lack of cross-platform compatibility and is no longer necessary.

I would consider Java more of a general-purpose language because of it's portable nature. This gives it more flexibility in what environments it can be ran on and thus supports a larger variety of applications. From personal experience, the amount of libraries and framework supported by Java also gives it an advantage in this sense as it makes it very easy to, for example, prototype a server-side web application. Due to the JVM, it is a lot higher level and whilst this has more of an overhead, it does provide useful features such as garbage collection. I think Java's abstraction of these details can be a double edged sword as it means developers learning Java as a first language would never experience things such as memory management or the compilation process.

Appendix A

Figure A.1: Typing in Java

```
//this will cause a compilation error
// as "hello" isn't of type int.
int a = 1;
a = a + "hello";
```

```
//this will cause a compilation error
//as the variable isn't declared.
a = 1;
```

Appendix B

Figure B.1: Implicit Type Conversion in C++

```
//this is allowed as they are both primitive,
//numerical types.
int i = 1;
short s;
s = i;
```

Figure B.2: Void pointer in C++

```
//Although this is allowed, it will have to be cast
//back to a non-void type before being used.
void *voidPointer;
int i = 5;
pVoid = &i;
```

Appendix C

Figure C.1: Pass-By-Value in Java

```
Car aCar = new Car("Ford", "Mustang");
foo(aCar);

public void foo(Car bCar) {
//This will change the name of the object aCar is
//pointing to as they both point to the same object.
        bCar.setModel("Mondeo");
//This will not change what object aCar is pointing to,
// only what object bCar points to.
        bCar = new Car("Vauxhall", "Astra");
}
```

Appendix D

Figure D.1: Pass-By-Value in C++

```
int a = 13;
printNumber(a);
void printNumber(int b)
{
        b=10;    // this wont effect a
}
```

Figure D.2: Pass-By-Reference in C++

```
int a = 13;
printNumber(a);
void printNumber(int &b)
{
        b=10; // this will make a=10
}
```

Appendix E

Figure E.1: new and delete operators in C++

```
//creates a new array in dynamic memory.
bool * boolArray;
boolArray = new bool[5];

//deletes array from dynamic memory, freeing space.
delete [] boolArray;
```

Appendix F

Figure F.1: Multi-threading in Java

```
Thread thread = new Thread(new RunnableA(n));
thread.setPriority(Thread.MAX_PRIORITY);
thread.start();
```

Figure F.2: Synchronized Method in Java

```
public class SynchronizedStudent {

        private String name = "Dom";

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

Figure F.3: Synchronized Block in Java

```
public class SynchronizedStudent {

        private String name = "Dom";

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

Appendix G

Figure G.1: Mutex in C++

```
string name = "Dom";

void setName(string str) {
        m.lock();
        name = str;
        m.unlock();
}
```

Appendix H

Figure H.1: Futures in Java

```
ExecutorService executorService =
Executors.newFixedThreadPool(4);

Future future = executorService.submit(new CallableA());

try {
        Integer result = future.get();
} catch (Exception e) {
        e.printStackTrace();
}
```

Appendix I

Figure I.1: Async Construct in C++

```
int i = 1;
int j = 100;
int add (int x) {
        i = i + x;
        return i;
}
std::future<int> future(std::async(add(100)));
int result = future.get();
```

Appendix J

Figure J.1: Error Handling in C++

```
int i = 1;
int add(int x) {
        i = i + x;
        if(i < 0) {
                throw "i LESS THAN 0";
        }
        return i;
}
try {
        int x = int(-5);
        cout << x << endl;
} catch (const char* msg) {
        cerr << msg << endl;
}
```

Bibliography

[1] "Albatross.". (N/A). History of C++. Available: www.cplusplus.com/info/history/. Last accessed 16/03/2016.

[2] Jon Byous. (2003). JAVA TECHNOLOGY: AN EARLY HISTOY. Available: https://www.santarosa.edu/~dpearson/mirrored_pages/java.sun.com/Java_Technology_-_An_early_history.pdf. Last accessed 16/03/2016.

[3] A. Bellaachia. (N/A). Object-Oriented Paradigm. Available: https://www.seas.gwu.edu/ bell/csci210/lectures/oop.pdf. Last accessed 16/03/2016.

[4] Premshree Pillai. (2004). Introduction to Static and Dynamic Typing. Available: http://www.sitepoint.com/typing-versus-dynamic-typing/. Last accessed 16/03/2016.

[5] N/A. (N/A). Parameter passing in Java - by reference or by value?. Available: http://www.yoda.arachsys.com/java/passing.html. Last accessed 16/03/2016.

[6] "ALEX". (2007). Passing arguments by reference. Available: http://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/. Last accessed 16/03/2016.

[7] N/A. (N/A). How Garbage Collection Really Works. Available: http://www.dynatrace.com/en/javabook/how-garbage-collection-works.html. Last accessed 16/03/2016.

[8] Andrei Milea. (N/A). Common Memory Management Problems in C++. Available: http://www.cprogramming.com/tutorial/c++_memory_problems.html. Last accessed 16/03/2016.

[9] N/A. (N/A). A garbage collector for C and C++. Available: http://hboehm.info/gc/. Last accessed 16/03/2016.

[10] N/A. (2004). The Compilation Process. Available: http://althing.cs.dartmouth.edu/local/www.acm.uiuc.edu/sigmil/RevEng/ch02.html#id2558687. Last accessed 16/03/2016.

[11] N/A. (N/A). Lesson: Concurrency. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/ Last accessed 16/03/2016.

[12] N/A. (N/A). Concurrency in C++11. Available: https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/labs/lab6/. Last accessed 16/03/2016.

[13] N/A. (N/A). Lesson: Exceptions. Available: https://docs.oracle.com/javase/tutorial/essential/exceptions/. Last accessed 16/03/2016.

[14] N/A. (N/A). Exceptions. Available: http://www.cplusplus.com/doc/tutorial/exceptions/. Last accessed 16/03/2016.