A comparison of C++ and Java.

Dominic Rathbone

March 8, 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 oreinted . 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 team known as the "Green Team" with the intentions of creating a modern alternative to C++ which was the most popular language at the time. The idea was spawned out of the green team's frustations with the C/C++ APIs that Sun had been using. It aimed to retain most of the feature set that C++ had whilst removing the unnecessary, outdated parts.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 and so 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 a variable, it has the obvious benefit of preventing potentially serious errors from occuring.

Although C++ is mostly statically typed, it can also 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).

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 reference that object but it is not the object itself (see Appendix C). [6]

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 ambersand (see Appendix D, figure 2) [7].

0.6 Memory Management

Memory can be split up into parts, mainly 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 method's scope. The later 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 [8].

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 laborous 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. Another technique is explicit garbage collection, although C++ lacks this implicitly, it can be implemented via libraries such as the "Boehm collector" library. [10].

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.

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 them inherently 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 accurately.

0.8 Concurrency

Concurrency can be explained via into the concept of multithreading. 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 multithreaded 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 that object. Only one thread can hold this monitor at a time so only it can change the data inside this object (See Appendix F, Figure 2).

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 multithreaded environment, C++ uses something called 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 timec(see Appendix G).

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 get method from to retrieve the result (see Appendix J). 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 K).

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 L). Although C++ is closer to the machin, it is not possible to set thread priority with the standard library.

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 FileNotFoundException. This can be detected at compile time if the file is not there and has to be dealt with before the program works. Unchecked exceptions are exceptions that unchecked at compile time as they occur at runtime. An example of this is ArrayIndexOutOfBoundsException as it is unchecked because it cannot be detected at compile time. 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 M).

Modern C++ also uses exceptions to handle errors but it doesn't have the notion of a checked exception. All exceptions are unchecked and are optionally caught. 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.

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 can use 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 compatability.

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.

Bibliography

[1] www.cplusplus.com/info/history/

[2] https://www.santarosa.edu/~dpearson/mirrored_pages/java.sun.com/Java_Technology_-_An_early_history.pdf

[3]

[4]

[5]

[6] http://www.yoda.arachsys.com/java/passing.html

[7] http://www.learncpp.com/cpp-tutorial/73-passing-arguments-by-reference/

[8] http://www.dynatrace.com/en/javabook/how-garbage-collection-works.html

[9]

[10] http://hboehm.info/gc/

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;
```