A comparison of C++ and Java.

Dominic Rathbone

March 7, 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 It aimed to retain most of the feature set that C++ had whilst removing the unnecessary, outdated parts.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 in both of the languages can be split up into to parts, the stack and the heap. The former is the memory which stores predetermined values such as 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 which are liable to changing over time.

Java can be considered memory-safe as you can not directly control how 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). Although C++ lacks implicit garbage collection, it can be implemented explicitly via libraries such as the "Boehm collector" library [10].

0.7 Compilation

0.8 Concurrency

Preventing the corruption of shared data between concurrently executing threads/ tasks. Synchronisation of threads/ tasks to prevent race conditions. Can the user influence the priority of threads/ tasks

0.9 Error Handling

0.10 Reflection

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;
```