C# Versus Java

Austin Klum

Two extremely well known and similar programming languages are C# and Java. Some history on the two goes as such. Java was first released in 1996 with the promise of “Write Once, Run Anywhere”. Java is compiled into Bytecode which is platform independent. From the bytecode the program is handled by the Java Virtual Machine (JVM) that will actually run the program into a more system dependent version. Java’s syntax is modeled after C and C++ in order to feel familiar and gain a larger following. Java added the use of garbage collection, the process of automatically deallocating memory, in contrast to the popular C and C++ that Java was modeled after. This was a big draw for some developers as memory management can be tedious. Others viewed this poorly as performance was decreased and allowed programmers to be more lazy with their coding. The latest version of Java Edition is Java SE 12 released in March 2019. Java has been extensively updating and amending to the Java language since it’s inception, adding new request features to help promote and keep Java as a main stay in the industry. While Java has been primarily an imperative language, the extension of Java has allowed for more paradigm such as functional programing.

C#, pronounced C-Sharp, was release in 2001 by Microsoft to be released with its new .NET program. The primary reason for C# creation was to create a Java like programming language that Microsoft had greater control over. The predecessor was Visual J++, which was Mircosoft’s own version of Java. While much of J++ conformed to the Java language specification, certain features were not implemented. J++ also extended different areas of the language for callbacks

and delegates. Eventually Microsoft decided it would be better to create its own language which would eventually become C#. C# is complied into a form of bytecode called Common Intermediate Language (CLI). The CLI will be ran on a virtual machine called the Common Language Runtime (CLR). This is very similar in regards to Java which also complies into bytecode and is ran on a virtual machine. C# is primarily used with Windows and Microsoft backed systems. C# is primarily an imperative language, but since its start has had some other abilities such as functional programming.

Both languages compile into a form of bytecode and run on a virtual machine. Much of the basic functionality is the same with garbage collection, exception handling, and other runtime-level security. The CLR garbage collector tends to be faster than the JVM's garbage collector. In a study conducted by Luis Henriques and Jorge Bernardino, "Our results show that, for deep deallocation, C#'s garbage collection system outperforms … Java by 5.26 milliseconds, which is surprising. We didn't expect C#'s garbage collector to outperform Java's by such extent". The CLR was designed to be language neutral and run primarily on Windows based device. While the VM was designed primarily for Java but was to be operating system independent. This is more of a board generalization as in recent years there has been greater push by both sides to pick up what the other lacks. Both the CLR and JVM use Just-In-Time (JIT) compilation, but they differ on how they go about it. Javacodegeeks.com says

> "The CLR compiles all MSIL code into machine code when it is called at runtime. The JVM uses a specialized performance engine called HotSpot to do JIT compilation of Java Bytecode into machine-readable code. This method only compiles the "hot spots" in the code that will actually be used."

Each strategy has its own pros and cons. While compiling everything at runtime is more costly at the start, over the long term of the program the execution time can be improved. While Java's HotSpot only compiles what is actually needed to be ran, if everything still needs to be ran there will be performance loss. The CLR was designed to be ran with instructions with more generic types and for applying parametric specializations on those types at runtime. This allows the CLR to recognize more types and be more flexible in nature. JVM has all value types fixed for now, but eventually plan on changing to something that mimics the CLR. With exceptions and error monitoring tools, the JVM has a wider array and more powerful tools that allow for better handling of debugging and monitoring.

One of the powerful tools that C# has that Java doesn't is Language-Integrated Query (LINQ). The Microsoft Docs explain LINQ as such: "Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language". LINQ is a declarative way of getting data from databases, lists, and other data structures. LINQ will automatically format the LINQ query into a format that the actual datasource can process.

```
// Specify the data source.
int[] scores = new int[] { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
```

This makes development easier as one can instantly access the data without having to write complicated loops, queries, or other processing functions. Java does have equivalent methods of doing this, but to much greater difficulty both in creation and understanding. The use of lambdas and streams in Java would accomplish the same task, but often times the lambda add more verbosity and complication, whereas the LINQ is query like and easy to understand.

Exceptions are another area where both Java and C# have similar functionality, but also go about some issues complete differently. Java has both unchecked and checked exceptions. Where C# only has unchecked exceptions. Unchecked exceptions are found during runtime. Often times an unexpected value or null point will cause these kinds of exceptions. Checked exceptions are checked at compile time. This requires the programmer to surround their code in try catch blocks or add a throws keyword expressing that somewhere else the exception will be handled. Anders Hejlsberg, lead C# architect said the following when asked about C#'s lack of checked exceptions. "I see two big issues with checked exceptions: scalability and versionability." When discussing the addition of a new exception that could be thrown from a hypothetical function that was modified, Hejlsberg said referring the programmers that will have to maintain that new addition to the function, "They're not going to handle any of these exceptions. There's a bottom level exception handler around their message loop. That handler is just going to bring up a dialog that says what went wrong and continue.". Hejlsberg also went on discussing how since you can just add a throws declaration in Java once you start to build up really large systems often times you'll end up having to try catch on numerous different exceptions. What ends up happening is they'll handle a case or two and then send the rest of the exceptions into a general catch and display the error message from there. Basically, the biggest reason why not to add checked exceptions is a lot of times there is nothing to be done to handle an exception other than just stopping the process.

C# also supports the use of Structs. Structs are similar to classes but are more lightweight. Structs in C# can have constructors, constants, methods, and many other features found in

classes. The main difference between structs and classes in C# is one is value-type while the other is reference-type. Structs are recommended over classes when the type is smaller than 16 bytes, immutable, short-lived, and not frequently boxed. When a struct has these conditions it is more efficient as it'll end up using stack space rather than heap space. Java does not have anything of this nature, instead opting for always using classes. This adds to more boilerplate code that some would argue adds bloat and lessens readability to the program. Often times structs are useful for quickly encapsulating data for when a full blown class is overkill for the work required. Structs also automatically have accessors and mutators by definition provided by C#.

```
public struct Point {
  public int X;
  public int Y;

  public Point(int X, int Y) {
      this.X = X;
      this.Y = Y;
  }

  public override string ToString(){
      return ($"({X}, {Y})");
  }
}
```

```
public class Point {

    public int X;
    public int Y;

    public Point(int X, int Y) {
        this.X = X;
        this.Y = Y;
    }

    @Override
    public String toString() {
        return ("(" + X + "," + Y + ")");
    }
}
```

Some other minor differences can also be found. One such difference is the decimal data type. In C# decimal has 128 bits for greater precision and accuracy. This is useful for financial applications were decimal representation is vital. Java on the other hand does not guarantee the accuracy of its double and float types. Because of this if one wants to represent currency, they will have to use either the Currency class or BigDecimal. This just adds more code to represent the underlying structure. Another minor difference between the two is the behavior of switch statements. In C#, case labels do not fall through to the next case label. Where in Java

once a case is true, it will fall through to the next case until hitting a break. C# also allows for switching on a string, where Java does not have this functionality. Another difference is the use of C#'s properties which take the place of getters and setters in Java. Benji Weber briefly mentions some of the differences between C# and Java for accessors and mutators, "C# properties, replace separate getters and setter methods that often clutter Java code. They allow the getters/setters to be combined together, allow passing of a property reference as a whole." He goes on to argue how this is a beneficial feature C# has but warns of some of the pitfalls and dangers of going overboard and killing readability.

Java

```
private int mSize;

public int getSize() { return mSize; }
public void setSize(int value) {
  if (value < 0)
    mSize = 0;
  else
    mSize = value;
}
```

C#

```
private int mSize;

public int Size {
  get { return mSize; }
  set {
    if (value < 0)
      mSize = 0;
    else
      mSize = value;
  }
}
```

C# and Java have had a long history together. Often times what one implements will be implemented by other in the next version. This way of leapfrogging off each other's ideas has allowed for two to produce very similar functionality. There are still of course many differences that separate the two. The VM's that run their bytecode will handle certain scenarios differently. Much of the syntax is the same, with C# trying to have easier to use and less verbose coding practices. This is expected as C# was released after Java and could benefit from Java's mistakes. A large market of business programming is captured by these two giants. I expect their influence and differences to have a large impact on the culture of software development for a long time yet. It will be interesting to see what comes next.

Works Cited

Soroker, Tali. “CLR vs JVM: How the Battle Between .NET and Java Extends to The VM-Level.” Java Code Geeks, 25 Jan. 2018, www.javacodegeeks.com/2018/02/clr-vs-jvm-battle-net-java-extends-vm-level.html.

“Language-Integrated Query (LINQ) (C#).” Language-Integrated Query (LINQ) (C#) | Microsoft Docs, docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

Venners, Bill, and Bruce Eckel. “Remaining Neutral on Checked Exceptions.” The Trouble with Checked Exceptions, www.artima.com/intv/handcuffs.html.

Henriques, Luis, and Jorge Bernardino. “Performance of Memory Deallocation in C++ , C# and Java.” Association for Information Systems, 2018, aisel.aisnet.org/cgi/viewcontent.cgi?article=1009&context=capsi2018.

Weber, Benji. “C# Style Properties in Java.” Benji's Blog -, benjiweber.co.uk/blog/2014/03/20/c-style-properties-in-java/.