CI346 – Programming Languages Comparison

Programmers today have more programming languages at their disposal than ever before and with so much choice comes the importance of knowing what sets these languages apart from one another. There are many important aspects to consider when comparing programming languages; some of which affect how the code is written, such as the paradigm of the language, and others that are less noticeable until the code is run such as whether it is a compiled or an interpreted language. Despite their names, Java and JavaScript are two languages that offer an abundance of differences when compared.

The first difference to look at between the two languages is how the code gets executed. Defining a language as either interpreted or compiled challenging as all programming languages can theoretically be executed using both methods. However, languages are typically implemented primarily using one of the execution methods which means that although it may not be a core feature of the language, it is still a valid point on which languages can be compared.

JavaScript is primarily an interpreted language meaning code is executed one line at a time by an interpreter program. One of the biggest advantages of being an interpreted language is platform independence, programs can easily be run on any platform regardless of hardware as the program is being executed by the interpreter rather than directly by the CPU. Being platform independent means that programs written in an interpreted language can easily be distributed which makes JavaScript an ideal language for web-based applications.

Working in an interpreted language can also lead to significantly shorter debug cycles as code changes can be tested immediately without having to wait for a potentially lengthy compilation stage. This can even go as far as modifying code whilst the program execution is halted at a break point, allowing programmers to quickly see the effect the code change will have on the program.

Java is harder to define in the sense of whether it is interpreted or compiled as it falls into both categories even more so than most languages. Whereas other compiled languages will compile code directly to machine code, Java code is firstly compiled into Java Bytecode which is in turn interpreted by the Java Virtual Machine (JVM). Because of this, Java has the same benefit of platform independence as JavaScript does, providing the platform it is running on is Java enabled.

The execution method of a language also ties in to how it is type checked. Java is a strong and statically typed language meaning that the type of a variable is known at compile time and once variables have been declared as a certain type, they cannot be assigned a value of a different (see appendix [1]). The advantages associated with being a statically typed language are that type errors, such as trying to use the ‘+’ operator between an integer and a string, get caught at compile time rather than run time. This leads to a more efficient development cycle as less time is spent tracking down and fixing type errors. Statically typed languages also have the advantage of being compiled into better optimised machine code that can be executed more quickly as the compiler already knows the data type of all variables being used.

The spectrum from weakly to strongly typed languages is completely separate from that of statically to dynamically typed languages. JavaScript lies on the opposite ends of both of these spectrums to Java, making it a weak dynamically typed language. It is also possible to have weak statically typed languages and strong dynamically typed languages. Being dynamically typed does not mean that variables in JavaScript do not have a type, it simply means that the type of any given variable is not checked until the code is executed. Once the code is executed, the variable is assigned a type based on the data it is storing. This allows variables in JavaScript to be initialised as one data type and then later assigned a value of a different data type. As it is also a weakly typed language it is possible in JavaScript to perform operations such as concatenating an integer and a string without the need to convert data types beforehand. This is known as implicit coercion as one of the variables is automatically converted to the same type as the other variable to make the operation valid. While there are obvious advantages to implicit coercion it can also lead to some unexpected errors that often prove extremely challenging to track down while debugging. A good example of this is that adding a variable of type string to a variable of type int results in the int being converted to a string and then the two strings being concatenated together to form a new string as the ‘+’ operator is used for concatenation in strings and addition in integers. However, as the function of the ‘-‘ operator is not defined for strings, subtracting a string from an integer results in the string being converted to an integer and a mathematical subtraction being performed on the two integers. This behaviour can be seen in appendix [2].

Errors caused by implicit coercion can be avoided either by always using explicit coercion , that is manually converting variables to the correct type before performing any operations on them, or by utilising the ‘===’ operator when comparing values or the typeof keyword to check the type of a variable before performing any operation on it. It can also be important to be able to check whether an object is an instance of a certain class before trying to call one of its functions, this can be achieved using the instanceof keyword as in appendix [3].

Class declarations were added into JavaScript with the release of ECMAScript 2015 (initially known as ECMAScript 6 or just ES6) in June of 2015 (En.wikipedia.org, 2020). Despite the addition of the class keyword in ECMAScript 2015, JavaScript still primarily uses prototype-based object orientation, classes “are primarily syntactical sugar over JavaScript’s existing prototype-based inheritance” (Parsy, 2018). An example of which can be seen in appendix [4].

Prototype-based object orientation differs from standard object-oriented languages due to the fact that classes are not used, and objects simply inherit from other objects. Instead of using a classic class hierarchy, JavaScript objects can be created and inherit from an existing prototype function. The newly created object is then free to call any of the existing methods from the prototype function or even create its own methods that are only available to itself. Similarly, a new prototype function can be created and then call the constructor method of an existing prototype function essentially inheriting the methods and member variables of the original function. This method of inheritance works by creating links between the *prototype* property of functions and the *\_\_proto\_\_* property of objects. This internal linkage means that, unlike in traditional object-oriented languages where behaviours and properties are copied in order to create an instance of a class, instances of JavaScript protype functions have their behaviour directly linked to their parent’s behaviour. By doing this, JavaScript programs reduce the amount of memory they use as the behaviour of all child objects simply references that behaviour in their parent object rather than having their own individual copies of that behaviour.

Although JavaScript primarily makes use of the object-oriented paradigm, it also allows programmers to write in the functional paradigm. Functional programming is declarative rather than imperative, meaning that code is written to declare what should be done without being specific about how it should be done as would be the case in an imperative language. This is achieved through the use of closures, first class functions, and lambda expressions. Functional languages also tend to focus on the use of recursion rather than iteration. Functions written in the functional paradigm have referential transparency as well as no side effects meaning that any call to a pure function using the same arguments will yield the same result every time. This is because values in functional programming are immutable meaning they are not affected by user inputs or the state of the file system. Isolating IO functions from pure functions makes it easier to find any bugs associated with IO operations.

Appendix 5 shows a comparison between using an imperative approach to summing the values of an array by iterating over each element and using a declarative approach by calling the reduce function on the array. In the imperative approach, the variable storing the result is mutable and its value gets changed after each iteration of the loop. The declarative approach reduces the array to a single value by summing its elements and therefore doesn’t change the state of any defined variables in the process.

In JavaScript, functions are treated as first class objects which means that they are able to be stored in variables, objects, and arrays; passed as an argument of a different function or returned by a different function. Anonymous functions can either be written in standard function notation for example:

[1, 2, 3].map( function (x) { return x \* x} );

Or as a lambda expression:

[1, 2, 3].map( x => x \* x );

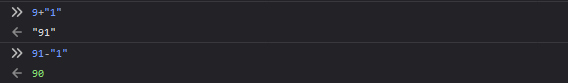
Both of these examples yield the same result, a new array containing the values of the original array squared and do so using the same declarative approach. However, the second example using a lambda expression, commonly known as a “fat-arrow” function, is more succinct as it omits the function and return statements.

Like JavaScript, Java is also a multi paradigm language that is primarily used in an object-oriented style. Java uses a more traditional approach to classes and inheritance, an example of which can be seen in appendix [6].

**Appendix A**

[1] – (Bhatnagar, 2018)

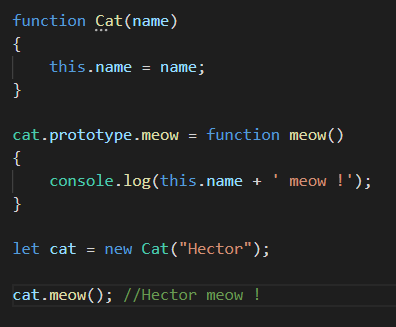
int data;  
data = 50;  
data = “Hello World!”; // causes a compilation error

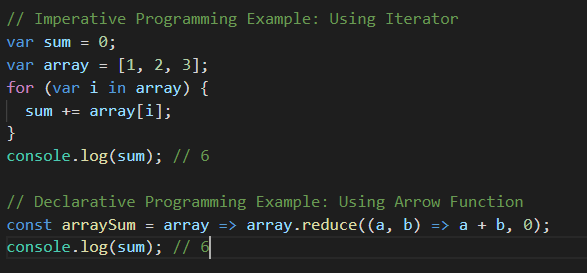
[2]

[3] - (Ng, 2019)

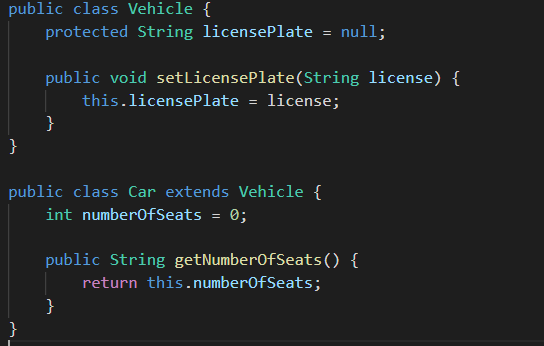


[4] - (Parsy, 2018)



[5] - (Austin, 2019)

[6] - (Jenkov, 2015)



**References**

Bhatnagar, M. (2018). *Magic lies here - Statically vs Dynamically Typed Languages*. [online] Medium. Available at: <https://android.jlelse.eu/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b>.

Ng, M. (2019). *Avoiding Weird Javascript Behaviour (true + true === 2 but true !== 1) - DEV Community 👩‍💻👨‍💻*. [online] Dev.to. Available at: <https://dev.to/neutrino2211/avoiding-weird-javascript-behaviour-true--true--2-but-true--1-pn9>.

Austin, D. (2019). *What Are JavaScript Programming Paradigms?*. [online] Medium. Available at: https://medium.com/javascript-in-plain-english/what-are-javascript-programming-paradigms-3ef0f576dfdb.

En.wikipedia.org. (2020). *ECMAScript*. [online] Available at: https://en.wikipedia.org/wiki/ECMAScript#10th\_Edition\_-\_ECMAScript\_2019 [Accessed 15 Jan. 2020].

Parsy, V. (2018). *Javascript : Prototype vs Class*. [online] Medium. Available at: <https://medium.com/@parsyval/javascript-prototype-vs-class-a7015d5473b>.

Jenkov, J. (2015). [online] Tutorials.jenkov.com. Available at: http://tutorials.jenkov.com/java/inheritance.html.