Part 1
Basic JavaScript

This short guide provides a brief introduction to the JavaScript Programming Language. It lays out the very fundamentals of the language to get you started with it. After going through this guide, you should be able to write simple programs using if-else statements, loops, functions etc. However, a fuller understanding of the language requires and broader introduction and more practice. For this purpose, I have provided links to some good resources throughout the guide and in the References section. You are encouraged to explore these recourses to enhance your skills.

This guide assumes knowledge of a programming language such as C++ or Python.

1. Programming Environment

JavaScript requires a JavaScript engine to execute. In the early days, JavaScript was used only to create dynamic web pages. JavaScript was written within a web page's HTML and would execute when the page was loaded. In that case, the browser's JavaScript engine would execute the *script*. However, nowadays, JavaScript is not only executed by browsers on the client side but can be executed on the server side or on any machine with a JavaScript engine installed on it. Some popular JavaScript engines are V8 (used in Chrome, Opera and Edge), and SpiderMonkey (used in FireFox), etc.

In this guide, we will use something more than JavaScript engine to develop and execute programs. We will use Node.js which is a JavaScript runtime built on the V8 JavaScript engine. A *runtime* – also known as a runtime system, or runtime environment – is a sub-system that exists both in the computer where a program is created, as well as in the computers where the program is intended to be run. Node.js is an open-source, cross-platform JavaScript runtime that executes JavaScript code outside a web browser.

The *Node.js on AWS* guide, provided separately, describes how to install the Node.js environment on the server side (the focus being that you will be writing client-server programs in JavaScript at some stage). However, in the current guide we will be writing and running programs on our local machine and our purpose will be to introduce the basics of JavaScript.

As a beginner, it is important that you make the following distinction between JavaScript and Node.js:
JavaScript is a programming language that runs in any browser JavaScript Engine. Whereas Node.js is an interpreter or running environment for a JavaScript programming language that holds many excesses, it requires libraries that can easily be accessed from JavaScript for better use.
The present guide does not discuss these excesses of Node.js and is very much restricted to simple JavaScript; except for some basic Node.js components. However, the set up described here will allow you to use any additional Node.js features which suit your purposes. In particular, you might want to write and test Node.js code on your local machine (using localhost as your server) before porting that code to your server on your AWS ec2 instance.

At this point, please take the following two steps:

(i) Install the Node.js runtime
(ii) Install Visual Studio Code

Sarim Baig
May 2022

Visual Studio Code (VS Code) is the code editor we will be using in this guide. It has support for the JavaScript language out-of-the-box as well as Node.js debugging. However, to run a Node.js application, you will need to install the Node.js runtime on your machine as indicated above.

2. The *Hello World* Program

Start VS code and create a new file called *HelloWorld.js* with the following code:

```
console.log("Hello World!")
```

- When you save the file, VS code will ask you to select a language. Select JavaScript.
- When you go to the *Run* menu and try *Run Without Debugging*, VS code will prompt you select an environment. Select Node.js.

You should see 'Hello World!' in the *Debug Console* when the program is executed.

There are different methods for output in JavaScript. For example, in the guide JavaScript from HTML, provided separately, you will see that when JavaScript is run from within a webpage's HTML, a method called document.write can be used to display text or even more HTML on the browser's window.

Node.js files can have either a .js, .node, *or* .json *file extension. The* .node *extension assumes that the file is a compiled binary, not a text file containing JavaScript. Therefore we stick to the .js extension.*

3. Variables in JavaScript

Execute the following code in VS Code to see what it does:

```
let num1=4, num2=5;
console.log("Sum: ", num1+num2);
```

There are three ways in which data is declared in JavaScript.

These are by using one of the keywords from **let**, **var** and **const**.

The difference between **let** and **var** is in how variables are scoped. When a variable is declared using *let* its scope is limited to the block – such as a function, if statement, or for loop -- in which it's declared. Using **var** the score of the variable is global if the variable is declared outside any function. If a variable is declared in a function using **var**, its scope spans that function and any nested blocks (including nested functions, as is possible in JavaScript)

This code should give an undefined reference error

```
{
    let num1=4, num2=5;
    //local score
    console.log("Sum: ", num1+num2)
}

console.log("Another sum: ", num1+num2);
```

This code should work just fine

```
{
    var num1=4, num2=5;
    //global scope
    console.log("Sum: ", num1+num2)
}

console.log("Another sum: ", num1+num2);
```

Variables declared with **const** are simply constant and cannot change their values. Their scoping follows the same rules as **let** variables.

> *As a rule, always declare variables with const. If you think the value of the variable can change, use let. If you're writing code for older browsers (let was only introduced in 2015) use var.*

If a variable is not initialized, it contains the value *undefined* by default, which has no type associated with it.

4. Data types in JavaScript

There are **7 primitive types** in JavaScript: string, number, bigint, boolean, symbol, null and undefined. However, as you can saw in the code above, in JavaScript you do not need to explicitly specify types when declaring variable unlike C++. This is because the latter is a strongly typed language and the former is a weakly typed language. You may read more about the differences between strongly and weakly typed languages in this Wikipedia article.

Execute the following piece of code to see the versatility of JavaScript types:

```
let num = 5, username = "s.baig", domain = "@imperial.ac.uk", type = true;
console.log(username+domain)
```

Even the following mixed-type concatenation works.

```
let num = 5, username = "s.baig", domain = "@imperial.ac.uk", type = true;
console.log(username+num+domain)
```

This indicates that in JavaScript you can apply 'object-oriented' operations on primitive data types. JavaScript makes this possible by creating temporary wrapper objects around primitive types with methods, such as the + operator in the above example, to facilitate the developer.

Other than primitive types, **objects** can be created easily:

```
let person = {
    name: "Sarim Baig",
    email: "s.baig@imperial.ac.uk",
    numOfModules: 4
};
console.log(person);
console.log(person.email);
```

Similarly, arrays can be created as follows:

```
let persons = ["Sarim", "Max", "Harry"];
console.log(persons);
console.log(persons[0]);
```

Get more information on data types by the following the tutorial on w3schools here.

5. **If-else statements and loops**

The following code is fairly self-descriptive and shows how if-else statements and for loops work in JavaScript, which is not too different from how they work in C++.

The program counts the number of elements in the array a between the values -10 and 10 both inclusive.

```
let a = [8,2,3,-1,4,12,33,21,-12,-5,20, 101, -91];
let count = 0;
for (let i=0; i<a.length; i++){
    if(a[i]>=-10 && a[i]<=10){
        count++;
    }
}
console.log("Number of elements between -10 and 10: ", count);
```

As you can see, the syntax is very close to C++. Notice the use of the length property of the array. Following is an extended example using **else if** and **else**.

```
let a = [8,2,3,-1,0,12,33,21,-12,-5,20, 101, -91];

let count = 0;
let zeros = 0;
for (let i=0; i<a.length; i++){
    if(a[i]>0){
        count++;
    }else if(a[i]<0)
    {
        count--;
    }else
    {
        zeros++;
    }
}
console.log("Difference between the number of +ves and -ves ", count);
console.log("Number of zeros ", zeros);
```

Moreover, JavaScript also supports **For in** and **For of** loops. The following examples demonstrate their use. **For in** traverses through the *indices* of the array whereas **For of** access the elements of the array.

Using **For in**

```
let a = [8,2,3,-1,4,12,33,21,-12,-5,20, 101, -91];
let count = 0;
for (let i in a){
    if(a[i]>=-10 && a[i]<=10){
        count++;
    }
```

```
}
console.log("Number of elements between -10 and 10: ", count);
```

Using **For of**

```
let a = [8,2,3,-1,4,12,33,21,-12,-5,20, 101, -91];
let count = 0;
for (let num of a){
    if(num>=-10 && num<=10){
        count++;
    }
}
console.log("Number of elements between -10 and 10: ", count);
```

The **while** loop is very similar to the while loop in C++

```

let a = [8,2,3,-1,4,12,33,21,-12,-5,20, 101, -91];
let count = 0;
let i = 0;
while (i<a.length){
    if(a[i]>=-10 && a[i]<=10){
        count++;
    }
    i++;
}
console.log("Number of elements between -10 and 10: ", count);
```

6. Functions in JavaScript

Following is the **general template** of a JavaScript function:

```
function example_function(param1, param2, ..., paramn){
    //function body...

    return ret_value;
}
```

Following is the code from section 5 now defined and called as a function:

```
function countAround0(a, range)
{
    let count = 0;
    let nrange = -range;
    for (let num of a){
        if(num>=nrange && num<=range){
            count++;
        }
    }
    return count;
}
let a = [8,2,3,-1,4,12,33,21,-12,-5,20, 101, -91];
```

```
range = 10;
console.log("Number of elements between ", -range, " and ", range, ": ", countAround0(a, 10));
```

Unlike C++, JavaScript allows you to write **nested functions**. This may be useful when a function is merely a helper to another function and does not make much sense outside its context. Following is an example:

```
function countAround0(a, range)
{
    function inRange(x, r){
        return (x>=-r && x<=r);
    }
    let count = 0;
    for (let num of a){
        if(inRange(num, range))
        {
            count++;
        }
    }
    return count;
}
let a = [8,2,3,-1,4,12,33,21,-12,-5,20, 101, -91];
range = 10;
console.log("Number of elements between ", -range, " and ", range, ": ", countAround0(a, 10));
```

We can write **functions as properties of objects**. See the following example:

```
let person = {
    name: "Sarim Baig",
    email: "s.baig@imperial.ac.uk",
    numOfModules: 4,
    introCard: function ()
    {
        return "Name: "+ this.name + "\n" + "Email: " + this.email + "\n";
    }
};
console.log(person.introCard());
```

A very nifty feature of JavaScript is its ability to treat **functions as data**. This means that we can assign functions to variables, and that has useful consequences. We can pass functions as parameters to other functions. We can return functions from other functions, assign them to variables and use these variables to call the returned function, etc. The following example shows such a use of functions:

```
function getAreaComputation(shape_type)
{
    if(shape_type=="circle"){
        return function(r){return Math.PI*r*r;}
    }

    if(shape_type=="rectangle")
```

```
    {
        return function(l,w){return l*w};
    }
}

area = getAreaComputation("circle");
console.log("Area of a circle with radius 3: ", area(3));
```

Check what happens when you pass a shape_type other than “circle” and “rectangle” to getAreaComputation. What exactly is returned by getAreaComputation in this case?

You might find the use of functions as data a very useful feature in web programming. Imagine, for example, a user clicking on a specific button on a web form being served a functionality accordingly. Or imagine a rover sending an error to the server being returned a specific error handler in return, etc.

Here is a good place to learn more about functions as data.

It will be useful at this stage to mention the use of ***‘this’* in the context of a JavaScript function**.

If the function resides in an object, *this* points to that object. If the function resides in the global context, *this* pointes to the global object. This global use of ‘this’ will make more sense when JavaScript functions are written within HTML. In that context, the global object will always be the current window. For now, the following example should give some insight into ‘this’ when it refers to the global object:

```
let ans = 0;
function summation(x, y)
{
    this.ans = x+y;
}
console.print("Golbal ans: ", ans);
```

The following example from *freecodecamp* show the use of *‘this’* in functions inside objects:

```
let blog = {
    name: 'Tapas',
    address: 'freecodecamp',
    message: function() {
        console.log("Name: ",this.name, "\nAddress: ", this.address);
    }
};
blog.message();
```

In this case, *‘this’* points to the object blog which contains the function message.

The following example from *freecodecamp* is more interesting:

```
function greeting(obj) {
    obj.logMessage = function() {
```

```
        console.log(this.name, "is", this.age, " years old!");
    }
};

const tom = {
    name: 'Tom',
    age: 7
};

const jerry = {
    name: 'jerry',
    age: 3
};

greeting(tom);
greeting(jerry);

tom.logMessage ();
jerry.logMessage ();
```

In this example, they have used a global function to assign a property (another function) to objects, and the 'this' has been reassigned to the appropriate object's context.

7. **User defined types: Classes**

Following is an example of a Class in JavaScript:

```
class Person{
    constructor(n, e, g){
        this.name=n;
        this.email=e;
        this.gender=g;
    }
    introCard()
    {
        return "Name: "+ this.name + "\n" + "Email: " + this.email + "\n";
    }
};
let person = new Person("Sarim Baig", "s.baig@imperial.ac.uk",'M');
//the line above uses the constructor of Person
console.log(person.introCard());
```

constructor is similar to the C++ constructor. However, its not named after the class named, but always called constructor. name, email and gender are the properties of the class created inside the constructor, using the 'this' operator. Also notice the member function introCard and the use of 'this' inside it.

Object Oriented Programming is not entirely the same in JavaScript as in languages like C++. In fact, JavaScript is not a class-based object-oriented language (but recently it has been given support to do that) but a prototype-based language. To understand the difference between these two approaches, and especially, if you wish to explore the object oriented

features of JavaScript in more detail, I recommend this starter tutorial from freecodecamp on How JavaScript Implements OOP.

Having said that, you can implement class-based inheritance in modern JavaScript in a quite similar way to languages like C++.

8. A note on memory management in JavaScript

In the last example above, we made our first use of the **new** operator. This lets us allocate memory for user-defined objects.

The memory management lifecycle of a variable, in any programming language, includes the following stages:

(i) Memory allocation
(ii) Reading and writing to/from the memory
(iii) Releasing the allocated memory

Stage (ii) is explicit in all languages, as it is part of your program's logic.

Stages (i) and (iii) are explicit in low-level languages, such as C++ -- recall the use of the new and delete operators in C++.

However, stages (i) and (iii) are mostly implicit in high-level languages, such as JavaScript. Specifically: the JavaScript engine takes care of allocation and deallocation for you for primitive datatypes by implementing automatic allocation and garbage collection.

For user defined types, however, allocation has to be explicit in JavaScript; but garbage collection is still automatic, so you do not need explicit delete.

In terms of automatic allocation (without the new operator): JavaScript uses both static (compile time, on the stack) and dynamic (run time, on the heap) allocation. Here is a good place to read about automatic allocation on heap and stack in detail. The 7 primitive types, since their sizes are known at compile time, are allocated space on the stack. Objects, such as the object person defined above, is allocated dynamically on the heap. This is because we may add new properties to (or remove properties from) an object at run time and its size might change. Similarly, arrays are also allocated dynamically on the heap.

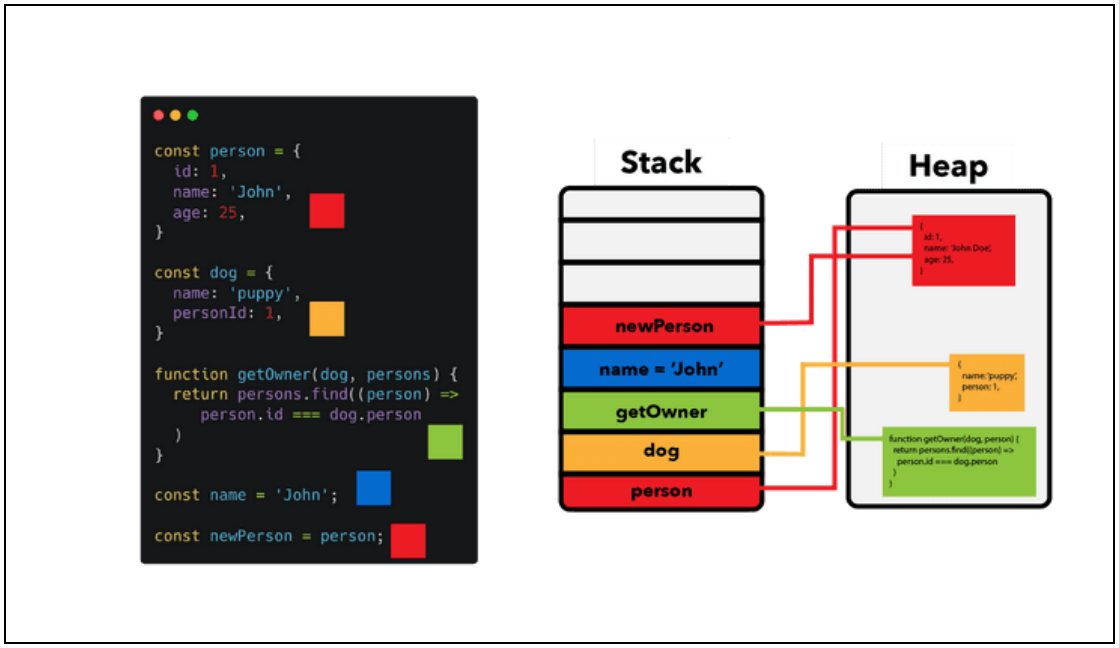
In short, then, JavaScript stores objects and functions in the heap. While primitive values and references are stored in the stack.

*In case of a user defined type, we need to make explicit use of the keyword **new** to specify the creation of a variable, according to our specified template, on the heap.*

9. References in JavaScript

This bit is important to avoid annoying bugs from entering your code.

Have a look at the following picture from Felix Gerschau's blog on JavaScript.



In this example of implicit allocation, objects and function are created on the heap; whereas primitive types and references are created on the stack. The important thing to notice is how the references named *person* and *newPerson* both point to the same copy of the object on the heap. What this indicates is that, by default, copying is shallow in JavaScript. Any future accesses to *person* and *newPerson* will access the same object. Therefore, it is important to keep the shallow nature of these copies in mind while coding in JavaScript to avoid unexpected behaviours and bugs in the code.

If at some stage you wish to make a deep copy, here is a comprehensive intro to deep copying in JavaScript.

References

1. The w3schools JavaScript tutorial
   https://www.w3schools.com/js/

2. The Modern JavaScript Tutorial
   https://javascript.info/intro

3. Node.js tutorial in Visual Studio Code
   https://code.visualstudio.com/docs/nodejs/nodejs-tutorial

4. The w3schools Node.js tutorial
   https://www.w3schools.com/nodejs/default.asp