# Programming in JavaScript

## Section 1: JavaScript Basics

### 1.1 Introduction

The HTML <script> element tag defines an inline and external JavaScript. Traditionally, we can also refer to scripts other than JavaScript e.g. PHP. In HTML5 the <script> tag assumes the script to be JavaScript.

Below is an example code for an inline JavaScript code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Inline JavaScript Example</title>
    </head>
    <body>
        <script>
            windows.alert('Hello, World');
        </script>
    </body>
</html>
```

**Note**: This code displays the string Hello, World! in the browser's alert window.

console.log('Hello, World!'); on the other hand will print to the browser's JavaScript console within the developer tools.

While it is possible to write scripts this way, we would typically want to make the script file separate from the HTML. There are a number of advantages of having an external JavaScript file such as it can speed up performance if multiple pages refer to the script file because the browser can cache request to the same file, it avoids code duplication, easy to use tools such as linters if the file is separate and easy to edit/maintain/reuse the code.

To create a external JavaScript file we would typically create a external script js directory and store all of our JavaScript files in this directory. The JavaScript file will have an extension of .js for example script.js and this file can be loaded into our HTML file using the <script src=""> tag pointing to the relative path from the .html file. bBelow is an example:

```
js > JS index.js
1    windows.alert('Hello, World');
```

```
<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>External JavaScript Example</title>
    </head>
    <body>
        <script src="./js/index.js"></script>
    </body>
</html>
```

**Disclaimer**: The <script> tag can be places in either the <head> or <body> element tags but for performance reasons when loading synchronous scripts it is often encouraged to place the script tag at the bottom of the <body> element. That way the page content loads and is parsed before waiting for the script to download and execute.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>External JavaScript Example</title>
    <script src="./js/index.js" async></script>
</head>
<body>

</body>
</html>
```

If the script tag is placed in the <header> element we can add the async attribute which will allow the ability to load external scripts asynchronously while the page content downloads.

To download the script while the page content is downloading and execute the script after the HTML has parsed we can use the defer attribute. This will behave and treated as if the script was added to the bottom of the <body> element tags.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>External JavaScript Example</title>
    <script src="./js/index.js" defer></script>
</head>
<body>

</body>
</html>
```

**Disclaimer**: Assuming the web server supports loading scripts in parallel we would potentially get a slight performance boost but it is recommended to stick to the <script> tag for external JavaScript files to be placed at the bottom of the <body> element tag.

---

## 1.2 Primitive Variables

A Variable is a reference in memory that defines a particular value. Variables can reference other variables and their values and the variable values can be changed if declared as mutable properties i.e. var and let.

There are different ways to declare variables. Variables names must start with a letter and contain no spaces (some special characters such as the underscore ( _ ) is allowed). Variables can contain numbers so long as it is not the first character in the name.

To assign a value to a variable we use the equal ( = ) sign. This will assign the value on the right of the equal sign to the variable name on the left of the equal sign.

We can use variables as placeholders for values, functions, equations and more. Variables can be set to reference another variable or the result of another variable.

```
var myVariable;

var myVariable2 = 20;
console.log(myVariable2);  // Prints 20 in the console

var myOtherVariable = myVariable2 + 10;
```

```
1   var myVariable;
2   myVariable = 20;
3   console.log(myVariable);  // Prints 20 in the console
4   myVariable = 10;
5   console.log(myVariable);  // Prints 10 in the console
```

**Important**: There is a difference between declaring and defining a variable. In the second example above, line 1 declares a variable telling the interpreter a variable exists. The second line defines the value of the variable i.e. to be 20. The value of the variable can be changed at any time by defining a new value.

JavaScript ES5 defines five primitive data types and one special type:
Number, String, Boolean, Null, Undefined and Object.

Functions are considered an Object data type and have the ability to be called. ES6 also adds another data type called Symbols.

```
var a = 1;
var b = 2;
console.log(a + b);  // Prints 3 in the console

var a = 'String A';
var b = 'String B';
console.log(a + b);  // Prints StringAStringB in the console

var a = '1';
var b = '2';
console.log(a + b);  // Prints 12 in the console
```

A number in JavaScript is a number regardless if it is an integer, long, float or double compared to other programming languages.

Strings are a sequence of characters strung together and is used to represent textual data.

We can use the loose equality operator ( == ) to compare two variables. We can also use other operators on variables which acts differently depending on the variable type. For example the addition ( + ) operator on number variables will add the two variables while using it on a string data type will concatenate the data into a concatenated string.

JavaScript is a loosely typed language which means that variables can be set to any data type without what type it needs to be first and it is allowed to change to a different type at any moment.

A boolean is a data type with two possible values i.e. true or false. When adding two booleans together the boolean is cast into the number 1 for true and 0 for false. In JavaScript type casting is the action of turning the types of one variable into another.

**Disclaimer**: Resolving different data types and type casting in JavaScript is very complex and a very common source of problems which is why the superset languages like TypeScript exists.

The strict equality operator ( === ) checks the equality not only in the value of the variable but also the data type itself. Therefore a number 1 is not the same as a string of ‘1’ because the data types are different.

Therefore, the loose equality operator will tell JavaScript to try to resolve the variable types before checking if they are equal to one another while the strict equality operator checks if the data types are the same and only then to perform the equality check.

```
var a = 12;
var b = '12';
console.log(a == b);  // Prints True in the console
console.log(a === b);  // Prints False in the console
```

**Note**: There are two ways to add comments in JavaScript. The first is the single line comment which is added by using two forward slashes ( // ) to a line of code and everything after the forward slashes are commented. The second is to use the forward slash and the start symbols ( /* */ ) and anything inside of this is commented out.

Null is the intentional absence of a value i.e. it means that a variable points to nothing. When a variable is implicitly absent of a value this makes the variable a data type of undefined. Therefore, if a variable has not been explicitly assigned a value it is considered

undefined. The most common place to see undefined variables are properties and objects that do not exist.

**Disclaimer**: While it is possible to declare a variable the value of undefined it is not recommended because the purpose of a undefined variable is to signify that the value has not been intentionally assigned anything. Explicitly assigning a variable to undefined goes against the spirit of this concept.

## 1.3 Special Variables

In JavaScript Objects is a type of value that has properties and a type. We can think of a property as a variable that belongs to an object. Basic Objects, Arrays and Functions are all types of (derive form) Objects.

**Disclaimer**: JavaScript is a Prototypal language and is not a classical Object Oriented language. This means that Objects inherit from other Objects as opposed to Objects derived from Classes that inherit from other Classes. ES6 classes are syntactical sugar.

There are different ways to define a basic object in JavaScript.

```
// Literal Constructor:
var myObject1 = {};
myObject1.someProperty = 'value'

// Object Constructor:
var myObject2 = new Object();
myObject2.someProperty = 'value'
```

```
var myObject1 = {
    someProperty: 'value',
    anotherProperty: 2
};
```

The Object and its property can be declared and defined at the same time.

To define multiple properties in the second example we can separate each property with a comma ( , ) as the separator. The colon ( : ) is used to set the property value when defining the properties and values when the object is declared.

```
var myObject = {
    someProperty: 'value',
    anotherProperty: 2,
    innerObject: {
        innerProperty: false
    }
};

console.log(myObject.anotherProperty);
console.log(myObject.innerObject.innerProperty);
```

```
var myObject = {
    someProperty: 'value',
    '20anotherProperty': 2,
    innerObject: {
        'inner Property': false
    }
};

console.log(myObject['20anotherProperty']);
console.log(myObject.innerObject['inner Property']);
```

Object properties can be defined as anything that a variable can including other Objects. To reference the inner object property we would use an extra dot operator.

Properties on objects are much more lenient on permitted names than raw variables. If we use quotes we can have properties that start with a numbers or have spaces, etc. To reference these properties we would need to reference it with a square brackets ( [ ] ) instead of the dot operator ( . ) as seen in the second example.

**Disclaimer**: We should stick to traditional naming conventions.

```
// Literal Constructor:
var myArray = ['a', 2, true];
console.log(myArray[0]);

// Array Constructor
var myArray = new Array('a', 2, true);
```

An array in JavaScript is a type of object that is used to store an ordered list of properties. Properties in an array are referred to as elements.

To access an element of the array we would use a square brackets with an index number to select one of the elements. Arrays in JavaScript uses zero based indexing whereby the first element has an index of 0. We cannot select an array using the dot operator as we can with Basic Objects.

Array elements can be of any type and the length of the array does not need to be specified when it is first declared. The array and all of its elements are mutable.

There is a special new Array constructor similar to the new Object constructor syntax as seen above for creating Basic Objects.

Arrays have a hand full of properties and functions for searching, mapping and filtering array objects. For example, the .length property returns a number of elements in the array.

```
var myArray = ['alpha', 'violet', 'romeo'];
console.log(myArray.length);  // Prints 3 in the console
console.log(myArray.sort());  // Prints ['alpha', 'romeo', 'violet'] in the console
```

## 1.4 Functions

A function is a type of Object that executes an inner script. It allows passing in arguments and returning a result or undefined if no results are specified. Common JavaScript functions are almost always created with the function operator keyword. The word after the function keyword is the name of the function.

```
function showSum(a, b) {
    console.log(a + b);
};
```

**Important Note**: It is a good naming convention to start a function with a verb. A good rule of thumb is to mentally split the words in the function name and see if it makes sense as to what the function is intended to do. In the above example the showSum functionality is to show the sum.

The function arguments are contained within the round brackets and are a list of variable names separated by commas ( , ). We can also leave this blank to be a parameterless function.

The function body is the line of code(s) that exist between the curly brackets ( { } ). The variable specified in the argument lists are available inside of the function body.

The function is not executed unless the function is called. To call a function write the function name followed by round brackets passing in any arguments (a.k.a parameters) as required.

```
showSum(1, 3);    // The function will output 4 in the console
```

```
function showSum(a, b) {
    return a + b;
};

var result = showSum(1, 3);
console.log(result); // Prints 4 in the console
```

To have a function return something we use the return statement. We can then store that return value in a variable which we can then output the value of the variable in the console.

Functions can return anything such as numbers, strings, objects and even other functions. A function does not always need to return data of the same type either but should avoid having functions that returns different types of data as it can lead to confusing issues.

The return keyword specifies the end of the function and will jump out of the function code block returning back to the code/line that called the function. The code after the return line will not be executed.

```
function showSum(a, b) {
    return a + b;
    console.log('Not executed');
};
```

```
var showSum = function (a, b) {
    return a + b;
};
```

An alternative way to define functions is to specify the name as a variable prior to the function operator. However, there is a very slight different to this approach.

When the JavaScript engine in the browser parses the script it does an initial pass to find all functions defined with the function operator first. It later executes the code. Therefore, this means that we can call functions before they are defined in the code if we define functions in a particular way. The below examples codes demonstrate this on a fresh web page load:

```
var result = calculateSum(1, 3)
console.log('result is: ' + result)

function calculateSum (a, b) {
  return a + b
}
```

```
result is: 4
```

```
var result = calculateSum(1, 3)
console.log('result is: ' + result)

var calculateSum = function (a, b) {
  return a + b
}
```

```
▸ Uncaught TypeError: calculateSum is not a function
    at <anonymous>:1:14
```

We can define functions as properties on objects as well.

```
var calculateSum = function (a, b) {
    return a + b;
};

var myObject = {
    sum: calculateSum
};

var result = myObject.sum(1, 3);
console.log('The result is: ' + result); // Prints The result is 4 in the console
```

Alternatively, this can be defined more concisely.

```
var myObject = {
    calculateSum: function (a, b) {
        return a + b;
    }
};

var result = myObject.calculateSum(1, 3);
console.log('The result is: ' + result); // Prints The result is 4 in the console
```

Just to show that a function is in fact a type of Object the below example creates the same calculateSum function using the Function constructor operator:

```
var calculateSum = new Function ('a', 'b', 'return a + b');
var result = calculateSum(1, 3);
console.log('The result is: ' + result);  // Prints The result is 4 in the console
```

**Disclaimer**: This approach should be avoided at all cost because it is slower because the interpreter needs to evaluate a string as opposed to raw code, introduces to potential security issues because no failsafe are handled in the string body, cannot take advantage of the debugger tools and it is a lot harder to read. Fundamentally, this is how functions are constructed in JavaScript.