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.

1.5 Scope

Scope is the current context of execution. The context in which values and expressions are “visible” or can be referenced. Child scopes have access to parent scopes but not vice versa.

A variable declared in a function body will not be visible to the outer scope or context. A scope of a function is known as the closure in JavaScript. The outer scope cannot reference the inner scope but the inner scope can see what is in the outer scope.

```
function add2(number) {
    var two = 2;
    return number + 2;
};

console.log(two) // two is not defined in this scope
```

```
var two = 2;

function add2(number) {
    return number + 2;
};

console.log(two) // two is defined in this scope
```

Note: In the first example the console.log( ) will error because the two variable is not in the outer scope whereas in the second example it will return 2 because it is in the outer scope. The function can still reference the outer scope two variable because it is a child of the outer scope. In this example the outer scope is the global scope.

```
var two;

function add2(number) {
    two = 2;
    return number + 2;
};

console.log(two) // two is defined in this scope
```

Since the function has access to the outer scope variable, it can also set a value to the outer scope variable within the function itself as seen on the example to the left. This will continue to log 2 in the console because both the inner and outer scope have access to the variable in the outer context.

```
var two = 50;

function add2(number) {
    var two = 2;
    return number + two;
};

add2(1) // This will return 3 (1 + 2)
console.log(two); // This will print 50
```

Scope is important because the value of scope allows us to separate concerns and isolate logic inside an inner scope and not worry about affecting the outer scope (see example to the left).

The inner function can even declare and define a

variable that already exists in the outer scope and it will not affect what is in the outer scope. The arguments of the functions are considered variables inside the function scope and are not visible to the outside scope.

```
var amountToAdd = 50;

function add22(number) {
    var amountToAdd = 20;

    function add2(number) {
        var amountToAdd = 2;
        return number + amountToAdd; // number + 2
    };

    return add2(number) + amountToAdd; // number + 20
};

console.log('1 + 22 = ' + add22(1)); ........// returns 23
console.log('amountToAdd: ' + amountToAdd); // remains 50
```

Scope can also be nested. In the example to the left, the inner most function declares and defines a amountToAdd as 2 which does not affect the parent function scope which sets the value to 20 which does not affect the global scope which defines the amountToAdd as a value of 50.

The add2 function inside of the add22 function is not available to the outer (global) scope either.

```
var amountToAdd = 50;

function add22(number) {
    var amountToAdd = 20;

    return (function add2(number) {
        var amountToAdd = 2;
        return number + amountToAdd; // number + 2
    }) (number) + amountToAdd; ......// number + 20
};

console.log('1 + 22 = ' + add22(1)); ........// returns 23
console.log('amountToAdd: ' + amountToAdd); // remains 50
```

Defining functions within functions in JavaScript is very common and used to isolate variables and logic within certain scope. The add22 body can be re-written to execute the add2 function without defining it on a separate line. We do this by surrounding the function in round brackets and then adding the opening and closing brackets at the end.

```
var amountToAdd = 50;

function add22(number) {
    var amountToAdd = 20;

    return (function (number) {
        var amountToAdd = 2;
        return number + amountToAdd; // number + 2
    }) (number) + amountToAdd; ......// number + 20
};

console.log('1 + 22 = ' + add22(1)); ........// returns 23
console.log('amountToAdd: ' + amountToAdd); // remains 50
```

Since the function is being called and defined at the same time there is no need to name the inner add2 function. When a function has no name it is called an anonymous function or a function expression. Anonymous functions are useful when defining self-executing functions or when passing functions as arguments to another function.

```
var numericArray = [1, 28, -1, 4, -70];

numericArray.sort();
console.log(numericArray); ....// returns [-1, -70, 1, 28, 4]

function compareNumbers(numberA, numberB) {
    return numberA - numberB;
};
numericArray.sort(compareNumbers);
console.log(numericArray); ....// returns [-70, -1, 1, 4, 28]

numericArray.sort(function compareNumbers(numberA, numberB) {
    return numberA - numberB;
});
console.log(numericArray); ....// returns [-70, -1, 1, 4, 28]

numericArray.sort(function (numberA, numberB) {
    return numberA - numberB;
});
console.log(numericArray); ....// returns [-70, -1, 1, 4, 28]
```

For example, the default behaviour of the sort method is to sort based off of the string representation of each of the array value which more often than not will not be what we want to sort by. In the example sorting in this way returns the wrong number.

To sort in the correct order we need to add an optional argument to the sort function. This is a function that sorts two values passed in as arguments. If the function passed in returns a negative number then the value for the first argument is place earlier in the array and if the function returns zero the order is left unchanged and if the function returns a positive number the second argument value is placed earlier in the array.

Since we are using the compareNumbers function in the sort call there is no reason to declare it as a separate function we can either call it in the sort function as an argument either named or anonymous.

```
var myObject = {
    innerProperty: 'Hello',
    getInnerProperty: function() {
        return myObject.innerProperty;
    }
};

console.log(myObject.innerProperty);      // Returns Hello
console.log(myObject.getInnerProperty);   // Returns Hello
```

So where we have properties inside of an object we can do the following. However, there are some limitation with this approach because variables are just references to values in memory.

If we made a variable that references a function that returns the objects inner property and then change that property in the object. The original myObject value has not changed even though nothing is directly referencing it but because the function is returning whatever the current myObject.innerProperty value is it returns the new object innerProperty value.

```
var myObject = {
    innerProperty: 'Hello',
    getInnerProperty: function() {
        return myObject.innerProperty;
    }
};

var getInnerProperty = myObject.getInnerProperty;
myObject = {
    innerProperty: 'Something Different'
}

console.log(myObject.innerProperty);      // Returns Something Different
console.log(getInnerProperty());          // Returns Something Different
```

Even more rudimentary than that, what if we wanted to change the name of the myObject variable. We would need to change the name of the variable everywhere including within the object, which means the inner object context needs to know about the outside state. This is not good programming practice.

```
var diffName = {
    innerProperty: 'Hello',
    getInnerProperty: function() {
        return diffName.innerProperty;   // Need to change it here too!
    }
};

console.log(diffName.innerProperty);
console.log(diffName.getInnerProperty);
```

```
var diffName = {
    innerProperty: 'Hello',
    getInnerProperty: function() {
        return this.innerProperty;
    }
};

console.log(myObject.innerProperty);
console.log(getInnerProperty());
```

To reference the former object inner property without memorising the name or without risking the object reference changing from outside the object by replacing the inner name with the this keyword.

The this keyword is a reference to the current content of the scope. In the global scope the this keyword will reference the window object of the browser.

```
function getInnerProperty() {
    return this.innerProperty;
};

var diffName = {
    innerProperty: 'Hello',
    getInnerProperty: this.getInnerProperty
};

console.log(myObject.innerProperty);
console.log(getInnerProperty());
```

When the this keyword is used in the function defined inside of an object it references the object that contains the function. The this keyword applies even if the function is defined outside the object definition. That is the basics of the this keyword. There are a lot of quirks to how the this keyword works especially when involving object prototype and functions that can change the content the function is being called in such as bind, call and apply.

While encouraged to play around with the this keyword to familiarise with the concept it is recommended to use it sparingly and only when it is very clear what the intention is in. There are better ways to structure code so that the this keyword is not necessary. For example, if we were to restructure the code, we would make sure that the inner property are not directly accessible and the only way to get it is through the get function. This way the outside code can modify and change its behaviour.

```
var myObject = (function () {
    var innerProperty = 'Hello',

    return {
        getInnerProperty: function() {
            return innerProperty;
        }
    };
})();

console.log(myObject.innerProperty);      // undefined because it is not exposed
console.log(myObject.getInnerProperty()); // Returns Hello
```

In the example to the left, the result of a self-executing anonymous function that defines it's own scoped variables and returns an object accessing those variables.

Another common approach for those with an Object Oriented Programming (OOP) background is to define a named function that does not self execute and create an object with the new operator.

The function does not return an object, the this object is returned otherwise it returns the object specified as the return. So we can use the this keyword and have the same result as demonstrated in the below examples:

```
var myObjectConstructor = function() {
    var innerProperty = 'Hello';

    return {
        getInnerProperty: function() {
            return innerProperty;
        }
    };
};

var myObject = new myObjectConstructor();
myObject.innerProperty;      // Should return undefined
myObject.getInnerProperty(); // Returns Hello
```

```
var myObjectConstructor = function() {
    var innerProperty = 'Hello';

    this.getInnerProperty = function() {
        return innerProperty;
    }
};

var myObject = new myObjectConstructor();
myObject.innerProperty;      // Should return undefined
myObject.getInnerProperty(); // Returns Hello
```

Disclaimer: JavaScript is not fundamentally an OOP language but there are some use cases where it makes sense to use the new keyword.

This is an important topic and you would need to understand how both the scope and this keyword works before moving onto other topics of JavaScript fundamentals. To conclude the scope is a mechanism for concealing variable declarations within smaller contexts so that they cannot be affected by the outside code and the this keyword is a reference to whatever the current scope is.

1.6 If Statements

The if statement executes a block of code if the condition between the round brackets after the if keyword is evaluated as “Truthy”. In JavaScript a “Truthy” value is one that if casted into a boolean type it will be true.

A value of 0, '' (empty string), null or NaN (not a number) will evaluate to a “Falsey” value. The logical not operator (!) converts a value or expression to a boolean and inverts it. Inverting the inverted boolean is the same as casting a value into a boolean outright:

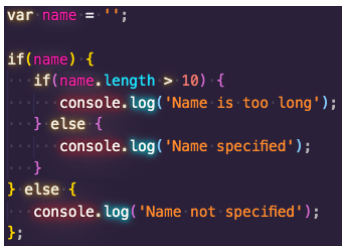
!false —> true
!true —> false
!0 —> true

!'hello' —> false
!!'hello' —> false

This is a very common practice in many JavaScript code bases.

The if statement construct can be followed by a else statement construct which will execute the block of code contained in the else statement if the condition is not met.

The if statement block of code will only run if the condition returns a “Truthy” value otherwise the else block of code will run instead. In the example to the left the JavaScript console will print 'Name not specified'.



If else statements can be nested as seen in the second example on the left. In this example if a name was not empty this will run the first block of code which starts with a nested if statement to check the length of the name variable. If the name is greater than 10 characters then the JavaScript console will print 'Name is too long' else it will print 'Name not specified'.

The nested if statement can be nested further using an else if statement. In the example to the right the else if statement will only be executed if the condition above it is not met i.e. the text must not be greater than 10 characters long which will then hit the else if statement to check that the text is not less than 2 characters long.

Logical operators within the if statement condition are used to help compare values in order to return a “Truthy” or “Falsey” values. The below tables provides the different logical operators available in JavaScript.

(Example) Logical Operator	Description
(a == b) ==	The equality operator. Checks if both values are equal to each other.
(a && b) &&	The AND operator. Checks if the former expression or value before it is truthy then return the second expression or value, else if it is not, then return the first expression or value.
(a || b) ||	The OR operator. Checks if the expression or value before it is truthy then return that value, else if it is not, then return the expression or value after it.
(a > b) >	The greater than operator. Check if the first value before it is greater than the second value after it.
(a < b) <	The less than operator. Check if the first value before it is smaller than the second value after it.

Any of these logical operators can be used outside of the if statements.

1.7 Loops

The while loop will loop until the condition is not met anymore. While loops tend to be more dangerous because if the condition is not written correctly the loop may never finish causing an infinite loop. The syntax for a while loop looks like the following:

while (condition) { code block; increment }

```
var number = 0;

while (number < 10) {
  console.log(number);
  number = number + 1;
};

console.log('Finished with number at ' + number);
```

The while loop uses the while keyword and the condition within the parentheses. The code block within the curly brackets is executed until the condition is no longer met. If the number variable did not increment by 1 during each loop would end up causing an infinite loop.

To increment a number we can use the increment operator (++) which will increment a number and returns the result. If the increment operator effects and mutates the value in memory. The increment operator returns the integer value before incrementing if the operator is placed after the integer and after incrementing if placed before the integer.

```
var x = 5;
var y = x++;    // x = 6 and y = 5

var x = 5;
var y = ++x;    //x = 6 and y = 6
```

```
var number = 0;

while (number < 10) {
  console.log(number);
  number++;
};

console.log('Finished with number at ' + number);
```

There is also a construct known as the do while loop where the code in the loop condition always execute at least once.

```
var number = 0;

do {
  console.log(number);
  number++;
} while(number < 10);

console.log('Finished with number at ' + number);
```

The do block of code will execute once and will continue to loop if the while condition is met. For example, if the number variable was 20 the do code will print 20 to the console and will no longer loop because 20 is greater than 10.

Typically when iterating through a loop there is a known starting point, a condition that specifies when to complete the loop, and an amount to increment by (usually by 1). The for loop is a short way to combine those steps into one long line. The syntax looks like the following:

for (start; condition; increment) { statement (a.k.a. code block) };

The start (initialisation) step is called before the loop begins and is typically used to declare variables. The condition is evaluated at each step in the loop and once it is no longer met the loop completes. The increment (final expression) is executed at the end of each loop step unless break is called.

```
for(var number = 0; number < 10; number++) {
    console.log(number);
};

console.log('Finished with number at ' + number);
```

```
var number = 0

for( number < 10; ) {
    var number: number
    number++
};

console.log('Finished with number at ' + number);
```

The start and increment can also be handled outside the parentheses as seen in the second example although the code looks much more shorter/cleaner in the first example.

The break statement tells JavaScript to exit out of the loop and is useful if the loop needs to check for multiple conditions.

```
for(var number = 0; number < 10; number++) {
    console.log(number);

    if(number == 5) {
        break;
    };
};

console.log('Finished with number at ' + number);
```

When breaking out of the loop the increment step does not execute. In the example to the left the number will remain at 5 and will not increment when it breaks out of the loop. Just like the return statement the break statement stops any further execution in the loop statement. If the console.log( ) was placed after the if statement it will not execute when breaking out of the loop.

The continue statement tells JavaScript to continue through the loop and ignore any code after it in the statement i.e. skip the rest of the loop statement for the current loop iteration.

```
for(var number = 0; number < 10; number++) {
    console.log(number);

    if(number == 5) {
        continue;
    };
};

console.log('Finished with number at ' + number);
```

In the example to the left, the code will console.log( ) number from 1 through to 9 but will skip logging number for when number is equal to 5 i.e. the JavaScript console will have the following output:
1 2 3 4 6 7 8 9 Finished at 10

The break and continue keywords works for both for and wile loops.