Programming in JavaScript
Section 3: ES6 and JavaScript

3.1 Const and Let Variables

Const and Let variables are a new feature that is introduced into ES6. The syntax is exactly the same as the var variable keyword; however, the behaviour are slightly different and will throw an error depending on the variable type used.

A const variable must be initialised when being declared i.e. we must assign it a value otherwise we would get an error.

A const or constant variable can store a value just like a variable. However, when declaring a const variable we must assign it a value (unlike var which does not require it to be initialised when declared).

```
var name;
name = "John";
```

```
const name; SyntaxError: Missing initializer in const declaration
```

```
const name = "John";
```

The var value can change its value anytime. The const variable on the other hand does not allow you to change the value of the constant variable and will result an error.

```
var name = "John";
name = "Mary";
```

```
const name = "John";
name = "Mary";          TypeError: Assignment to constant variable.
```

Therefore, when declaring a const variable we must assign it a value when it is declared and we cannot re-assign it a new value at any point in the code hence, as the name indicates, it has a constant value.

The let variable is almost the same as var but similar to const variables there is a difference between var and let variables. A var variable has two scopes a global and a function scope. The let variable has three scopes a global scope, function scope and a block scope.

When we declare a variable outside a function the variable is said to be declared in the global scope. This means any child functions of the code can access this global variable.

```
var global = "I am a global variable";

function sayMyName() {
    console.log(global);
};

sayMyName();                I am a global variable
console.log(global);        I am a global variable
```

A local scope (or a function scope) is a variable that is only available in the function and is not available outside in the global scope i.e. it is only local to the function it was declared in. The below code will throw a ReferenceError on console.log(local) because the global scope does not have access to the function variable outside the local scope. Therefore, the variable can only be manipulated inside of the function where it was declared.

```javascript
function sayMyName() {
    var local = "I am a local variable";
    console.log(local);
};

sayMyName();
console.log(local);          ReferenceError: local is not defined
```

The let variable functions exactly the same as the var variable i.e. it can be initialised at a later point in time and has both the global and function scope.

```javascript
let global;
global = "I am a global variable";

function sayMyName() {
    let local = "I am a local variable";
    console.log(global);
    console.log(local);
};

sayMyName();                 I am a global variable
console.log(global);         I am a local variable
// console.log(local);       I am a global variable
```

The console.log(local) code will throw a ReferenceError as seen above example for var. As we can see the let and var variables work exactly the same.

The only difference as mentioned above is that the let variable has a third scope which is called the block scope. In JavaScript a block is anything that starts and closes with curly brackets ({ }). The code contained in the curly braces is part of that block of code.

The let variable inside of a block can only be accessed in the block scope and cannot be accessed outside of the block i.e. in the global scope (as seen below this will return a ReferenceError).

```javascript
{
    let block = "I am a block scope variable";
    console.log(block);
};

console.log(block);          ReferenceError: block is not defined
```

When the block code completes its execution the let variable no longer exists which is why it gives the ReferenceError. This behaviour does not apply to var variables as demonstrated in the below example:

```javascript
{
    var block = "I am a block scope variable";
    console.log(block);
};
                             I am a block scope variable
console.log(block);          I am a block scope variable
```

Finally, it is important to note that a let and var variable cannot have the same name (no variables can be declared with the same name) and this will throw a SyntaxError to inform you that the variable name has already been declared. In the below example the variable name is called 'variable' and this will change to whatever the name of the variable is that has already been declared.

```
SyntaxError: Identifier 'variable' has already been declared
```

3.2 Block Scope Functions

When a function is created outside in the global scope and another function with the same name is created in a block scope, the new block scoped function will override the global scoped function.

Remember a block scope is anything that starts and ends with the curly brackets e.g. if statements, while, do while and for each loops.

```javascript
function foo() {
    return "b";
};

{
    function foo() {
        return "bar";
    };
    console.log(foo());
};

console.log(foo());
```

```javascript
{
    function foo() {
        return "bar";
    };
    console.log(foo());
};

function foo() {
    return "b";
};

console.log(foo());
```

In this example the foo( ) function in the block scope has overwritten the foot( ) function in the global scope and will print 'bar' in the console twice.

Note the same does not apply in the vice versa i.e. if we declare a function inside of the block scope first and then in the global scope declare a function with the same name, this will not override the block scope.

```
bar
bar
```

```javascript
{
    function foo() {
        return "foo";
    };

    {
        function foo() {
            return "bar";
        };

        console.log(foo());
    };

    console.log(foo());
};
```

When we have a block inside another block (nested blocks) the same behaviour occurs i.e. whereby the new function with the same name overrides the previous function as see in the example below:

The console.log( ) in the inner block returns 'bar' to the terminal/ console because the inner foo function overrides the outer block foo function.

However, the console.log( ) in the outer main block returns 'foo' in the terminal/console and is not overwritten by the inner block function.

```javascript
{
    let variable = "foo";
    function foo() {
        return variable;
    };

    {
        let variable = "bar";
        function foo() {
            return variable;
        };

        console.log(foo());
    };

    console.log(foo());
};
```

The reason for this behaviour is because the inner block code i.e. functions will only exist in the inner block execution and will override the outer block function with the same name. Once the inner block execution completes the inner function (code block) no longer exists. Therefore, the function in the outer block is now the only function that exists in the block execution.

Therefore, when working with inner block functions that have functions with the same name as the outer block functions we need to pay special attention to this behaviour of JavaScript. This behaviour also applies to variables within block scope.

```
bar
foo
```

Important Note: If the inner block 'variable' did not have the 'let' keyword then this will overwrite the outer blocks let variable and would end up printing 'bar' twice.

3.3 Optional (Default Value) Parameters

Before ES6 when a function had parameters it was required to pass in all the function parameters within the round brackets when calling the function.

```
function peopleToString(name, age) {
    return "Your name is: " + name + " and your are " + age + " years old.";
};

console.log(peopleToString("John", 35));        Your name is: John and your are 35 years old.
```

Now in ES6 if a parameter is not passed into a function the parameters will be replaced by the undefined data types as seen below:

```
console.log(peopleToString());Your name is: undefined and your are undefined years old.
```

ES6 also allows us to create optional parameters. To do this we simply assign a default value to the optional parameter(s) as demonstrated below. This will now use the default value unless the parameter is passed in which will override the default value.

```
function peopleToString(name, age = 30) {
    return "Your name is: " + name + " and your are " + age + " years old.";
};

console.log(peopleToString("John"));        Your name is: John and your are 30 years old.
console.log(peopleToString("John", 35));    Your name is: John and your are 35 years old.
```

3.4 Spread Operator

The spread operator (…) within the function parameters allows the function to receive an unlimited parameters. To have an unlimited parameters we would use the spread operator followed by the name of the parameter. This will act as an array where we can pass in any number of parameters.

```
function people(name, age = 45, ...family) {
    console.log(family);
};

people("John Doe", 35, "Julie Doe", "Barry Doe");
```

In this example, every other parameter values passed in after the age parameter will be added into the family array.
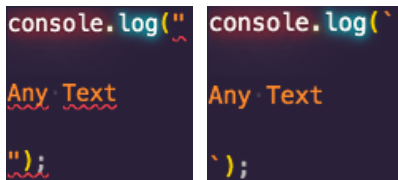
```
[ 'Julie Doe', 'Barry Doe' ]
```

This is a very powerful operator to make function parameters dynamic and have an unlimited amount of parameter that can be passed in based on the scenario. The spread operator in JavaScript is used with arrays. We can loop through the array to do some interesting things with the data.

```
function people(name, age = 45, ...family) {
    console.log("Your name is " + name + " and you are " + age + " years old. Your family members are:");
    for(var i = 0; i < family.length; i++ ) {
        console.log(family[i]);
    };
};

people("John Doe", 35, "Julie Doe", "Barry Doe");
```

```
Your name is John Doe and you are 35 years old. Your family members are:
Julie Doe
Barry Doe
```
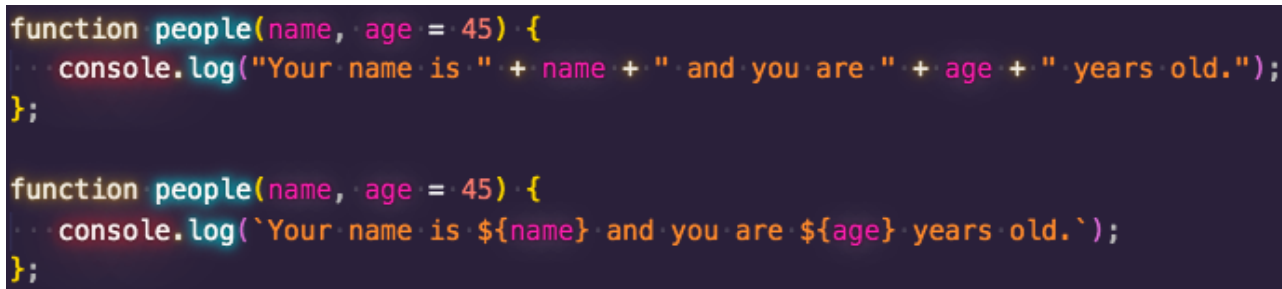
3.5 Template Literal

A template literal is a new powerful type of string which has many benefits over the regular string data type. To create a template literal we use double back ticks (``) instead of the single or double quotations.

A regular string does not allow you to add empty new lines which will end up as a SyntaxError. The template literal on the other hand does not have this problem and will print the new lines.

Strings require the add operator (+) to concatenate strings with variables. This syntax can look very unusual or unreadable for humans. Template literal overcomes this by using a special syntax of a dollar sing and curly brackets (${ }) which contains the variable name inside of the curly brackets in order to concatenate to the string. This results in a more easy syntax to write and read as demonstrated in the examples below:

```javascript
function people(name, age = 45) {
    console.log("Your name is " + name + " and you are " + age + " years old.");
};

function people(name, age = 45) {
    console.log(`Your name is ${name} and you are ${age} years old.`);
};
```

3.6 Binary Numbers

ES6 introduces the use of binary numbers inside of JavaScript. Binary numbers are part of the numerical binary system. In the real world we use the decimal numeric system which compose numbers with digits between 0 to 9. In the binary number system the numbers are composed of 0s and 1s.

To use a binary number in JavaScript we simply use 0b followed by the binary numbers for example the table below shows the binary number for 1 to 5 as an example:

Binary Number	Decimal Number Equivalent
0b001	1 (i.e. 1.0)
0b010	2 (i.e. 2.0)
0b011	3 (i.e. 3.0)
0b100	4 (i.e. 4.0)
0b101	5 (i.e. 5.0)

Mathematical operators can be used with binary numbers and we can also perform math operations even when mixing both the binary number system with the decimal number system.

```
console.log(0b101 * 0b010);    10
console.log(0b101 * 3);        15
```

Binary numbers must start with 0b followed by numbers that are between 0 and 1. Any number above 1 or below 0 will result in a SyntaxError.

3.7 ES6 New JSON Features

```
let name = "John Doe";
let age = 20;

people = {
    name: name,
    age: age
};
```

In ES5 to assign a variable to a JSON object property's value we required to write the object property name followed by a colon (:) followed by the variable name. Where the property name and the variable share the same name this causes a small code repetition.

```
let name = "John Doe";
let age = 20;

people = {
    name,
    age
};
```

ES6 introduces a shorthand whereby we can emit the variable name as the value if it shares the same name as the object property name. This results in a much more neat and readable code. The object properties takes the values from the variable defined above automatically via the variable name.

```
let animal = {
    type: "Dog",
    sound: function() {
        console.log("bark");
    }
};

animal.sound();
```

In ES5 to create a function within an object we had to create a property and then assign the value to a function. To call on the object function we would write the object name followed by the period (.) followed by the property name containing the function.

```
let animal = {
    type: "Dog",
    sound() {
        console.log("bark");
    }
};

animal.sound();
```

ES6 introduces a new way to work with functions inside of a JSON object which make it more simpler and readable i.e. you no longer need to declare a property name followed by the function( ) keyword. Instead you treat the property name as the function instead.

These are the two new features introduced in ES6 when working with JSON objects i.e. the object property shorthand and object functions syntax.

3.8 Destructuring

Destructuring is a new feature introduced in ES6 which makes it simpler to take the property values from an object and place them into variables. In ES5 to do this we would have had to use the syntax as seen in the below example:

```
var person = {
    name: "John Doe",
    age: 30,
    eyeColour: "Blue"
};

var name = person.name;
var age = person.age;
var eyeColour = person.eyeColour;

console.log(`Your name is ${name} and you are ${age} years old and have ${eyeColour} eyes.`);
```

The problem with this approach is that the code is much longer and should you decide to change the name of the property in the object then the property name would also needs updating for the variable values that references the object property. This can soon become very cumbersome in larger code base.

ES6 introduces destructuring which makes this much simpler to do in JavaScript with very little syntax. To destructure a object properties we start with the variable key word (i.e. either var, let or const) followed by curly brackets. Inside of the curly brackets the variable names are defined i.e. these are the properties we wish to destructure from the object. Finally, using the assign operator (=) we assign the object we wish to destructure the property values from to store in the variables we defined in the curly brackets.

```javascript
var person = {
    name: "John Doe",
    age: 30,
    eyeColour: "Blue"
};

var { name, age, eyeColour} = person;

console.log(`Your name is ${name} and you are ${age} years old and have ${eyeColour} eyes.`);
```

In the example above, the syntax tells JavaScript to go to the person object and extract the name, age and eyeColour property values and assign it to the (*var*) variables which are also named name, age and eyeColour.

Important Note: The variables on the left of the assign operator do not have to be in the same order as the properties defined in the object as long as the variable name is the same as the property name this will continue to work.

This is a much cleaner and simpler code for creating new variables and assigning it the values from an object. However, you would still have the same issue as before if you were to change the property name of a object but will be much easier to update the code than using the ES5 syntax.

3.9 Classes In ES6

ES6 introduces classes which are used for Object Oriented Programming (OOP). A classification is the process that groups properties and actions of a group of objects of animate/inanimate things. Therefore, classes are a way to categorise similar entities into groups. We can think that a class is a mould for an object.

We can define objects' attributes inside a class for example the People class can have attributes of name, age, height, weight, etc.

We can also define actions that the people class can perform for example walk( ), jump( ), run( ), etc. When we have functions in a class this is known as a method.

We can think of a class as an empty table and when we create a new object that uses a class we can think of it as filling the table with the information required for the class. Therefore, we can use classes to create custom data types.

To create a class in ES6 we need to use the class keyword followed by the class name. It is common practice to create a class name where the first character is uppercase. This will help distinguish a variable from a class object. We then use curly brackets to define the properties and methods of the class.

```
class Person {
    constructor() {
        this.name
        this.age
    };
};
```

Every class needs to have at least one property and to define these we require a special class method called constructor. The constructor is a special method of class within JavaScript.

To define the properties we use the this keyword followed by a period (.) and then the name of the property.

The constructor is a special method that gets executed when we create a new object from the class i.e. it is the first method that gets executed when we create a new object from the class. To create a new object from the class we would create a new variable and set its value using the new keyword followed by the class name. The new keyword creates a new copy of the class.

```
let john = new Person();
john.name = "John Doe";
john.age = 35;
console.log(john.name);
John Doe
```

To access a property from a class object we use the new object name followed by a period (.) and then the property name. To set a value to a class property on the new object we use the assign operator to assign the property a value.

We can think of a property as a variable inside of an object. We can create as many new objects from the Person class for each person and each person object will have a name and age property as seen in the above example.

This approach of defining object properties is not efficient and clean. Instead, we can define object properties values in the constructor using parameters. We can then pass in the property values when we instantiate a new object from the class as seen in the example to the right.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    };
};

let john = new Person("John Doe", 35);
console.log(john.name);
```

To create a method inside of a class is the same as creating a function within the class object i.e. we define a method name followed by a open and closing round brackets followed by curly brackets. To call a class method is the same as calling a property on an object, instead we call on the method as seen in the example to the right. Therefore, a method is simply a function inside of an object.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    };

    sayHello() {
        console.log("Hello");
    };
};

let john = new Person("John Doe", 35);
john.sayHello();
```

```
    sayHello(input) {
        console.log(input);
    };
};

let john = new Person("John Doe", 35);
john.sayHello("Hi");
```

We can define parameters inside of a method to make the methods more dynamic. When calling on the method on the object we can pass in arguments which makes the output dynamic.

Methods can also access object properties. The this keyword refers to the object. Therefore, when we call on the this keyword it is telling JavaScript to get the property of the object that called on the sayHello method.

```
    sayHello(input) {
        console.log(`${this.name} says: ${input}`);
    };
};

let john = new Person("John Doe", 35);
john.sayHello("Hi");
```

3.10 Static Methods

The static keyword makes a method in an object a static method. This means that we can access the method by referencing the class itself without having to create a new object of the class.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    };

    static onePlusTwo() {
        console.log(1 + 2);
    };
};

Person.onePlusTwo();
```

Without the static keyword this would have meant that we would have had to create a new object of the class and then access the method via the object.

In the example on the left, because the method has the static keyword we are able to call on the method from the class itself without having to create an object.

If we try to call on a method that is not defined as static from the class itself this will cause a Uncaught TypeError.

Therefore, a static method belongs to the class and not the object. If we try to access the static method from an object we would get an Uncaught TypeError error message as demonstrated below.

```
let john = new Person("John Doe", 35);
john.onePlusTwo();                         TypeError: john.onePlusTwo is not a function
```

3.11 Getters & Setters

Getters and Setters are special methods (similar to static methods) and allow us to manipulate properties and encapsulate them. Getters are used to get data from the object and Setters are used to define data in properties of the object.

To create a getter we would use the get keyword followed by method name that is the same name as a property existing on the class. A getter must always return some data.

To use a getter we simply would call the object followed by a period (.) followed by the property name as a variable. However, this would through a TypeError because in JavaScript for every getter there must also be a setter.

To create a setter we would use the set keyword followed by the method name that should also be the same name as the property existing on the class and the getter method. A set method requires an argument parameter for the method.

The setter is used to define the data (value) of the property while the getter is used to get the data from the property of the object.

The variable within the code block of the get and set methods cannot use the same property name as the method name because this would throw and Uncaught RangeError.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    };

    get age() {
        return this._age;
    };

    set age(age) {
        this._age = age;
    };
};

var john = new Person("John", 35);
console.log(john.age);
```

Every time you try to access a property in the object in the class this will access the setter and this behaviour makes the setter method recursive causing the RangeError.

To fix and good practice to this issue is to use the underscore (_) before the property name (i.e. you cannot have the same property name inside the getter and setter methods).

The getter and setter will now work without throwing the Uncaught RangeError.

To use a setter we simply would call the object followed by a period (.) followed by the property name as a variable. We would then assign a value.

```
var john = new Person("John", 35);
john.age = 20;
console.log(john.age);
```

In the example on the left we set the age property of the john object from 35 to 20 using the setter method.

Therefore, when we call on the property and use the assign operator JavaScript knows to call the setter method to assign a value to the object property while calling on the property without the assign operator will call the get method to return the value of the property. This is how we can use getters and setters with our class objects.

We have more power with getters and setters because we can manipulate the data and encapsulate the attributes/properties. Encapsulation involves providing methods that can be used to read from or write to the field rather than accessing the field directly.
This allows to manipulation of the input and output data as well as validate input to maintain data integrity.

```
set age(age) {
    if(age > 0 && age < 100) {
        this._age = age;
    } else {
        this.age = 10;
    };
};
```

In this example on the left the setter method would validate the data passed to ensures that the age property cannot be set to a number above 100 ensuring data integrity of the object.

This is the main power and purpose behind getters and setters.

3.12 Inheritance

Inheritance is a way to create classes that derive from other classes i.e. classes created based on other classes. One class that derives from another class will inherit all of the other classes properties and methods. Inheritance is a big topic in OOP.

Inheritance allows us to reuse code to help create another class that derives from another class without having to rewrite all of the property and methods again. This helps maintain the DRY principal of Don't Repeat Yourself.

The extends keyword allows a new class to derive properties and methods from another class. The class the new class derives from is specified after the extends keyword. This creates a parent child relationship between the two classes and the child class will automatically inherit all of its patent's class properties and methods.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    };

    get age() {
        return this._age;
    };

    set age(age) {
        if(age > 0 && age < 100) {
            this._age = age;
        } else {
            this.age = 10;
        };
    };
};

class Teacher extends Person {
    constructor(name, age, subject, classroomSize) {
        super(name, age);
        this.subject = subject;
        this.classroomSize = classroomSize;
    };
};

let john = new Teacher("John Doe", 40, "Mathematics", "Small");
```

When creating the constructor method inside of a inherited class we must use the super( ) constructor method inside of the constructor method otherwise we would run into a ReferenceError. The super( ) method execute the constructor of the parent class.

In the example on the left the super( ) method would execute the constructor method of the Person's class which is the parent class of the child Teacher class.

The super( ) method must be the first line of code within the child class constructor method. The Teacher class now inherits all of the properties and methods of the Person class as well as having its own properties and methods which it can access.

A class can inherit from the parent of another class which also inherits from another class causing a chain of inheritance. In the example on the below the MathsTeacher class inherits the properties and methods from both the Teacher class and the Person class.

```
class MathsTeacher extends Teacher {
    constructor(name, age, subject, classroomSize, favouriteMathTopic) {
        super(name, age, subject, classroomSize);
        this.favouriteMathTopic = favouriteMathTopic;
    };
};

let john = new MathsTeacher("John Doe", 40, "Mathematics", "Small", "Algebra");
```

As we can see inheritance is very powerful and allows us to adopt a DRY approach to our code and is also important to OOP.

3.13 Destructuring in Arrays

The code example to the right which demonstrates extracting each name within the array and storing the values in separate variables is considered bad coding because there are a lot of repetition in the code.

```
let arrayOfNames = ["John", "Mara", "Peter", "Abigail"];

let person1 = arrayOfNames[0];
let person2 = arrayOfNames[1];
let person3 = arrayOfNames[2];
let person4 = arrayOfNames[3];

console.log(person1, person2, person3, person4);
```

The ES6 destructuring helps us to achieve a much more cleaner code and can also be used in arrays. The destructuring operation is used to extract values from an object (as seen in 3.8) but it can also be used to extract values from an array.

To destructure an array we use the variable keyword and within square brackets ([ ]) we would define the variables we wish to create of the variable type and assign its value to the name of the array. The example on the right achieves the same result as the above code but in a much neater way.

```
let arrayOfNames = ["John", "Mara", "Peter", "Abigail"];

var [person1, person2, person3, person4] = arrayOfNames;

console.log(person1, person2, person3, person4);
```

The destructuring of arrays work in order of the array i.e. the first variable is assigned the value of the first item in the array, the second variable assigned the second value in the array and so on. We can omit some values from the

```
let arrayOfNames = ["John", "Mara", "Peter", "Abigail"];

var [person1, , person3, person4] = arrayOfNames;

console.log(person1, person3, person4);
```

array by leaving the variable empty and this will inform JavaScript when destructuring values as seen in the example on the left which omits the second array item.

3.14 Find Items in Arrays

In ES6 the .find( ) method allows us to find a item inside of an array. The .find( ) method takes in a function to call. In the below example this uses a ES6 arrow function which we will learn in more detail later on and should ignore for now. The return value from the find method should be stored in a variable.

```
var arrayOfNumbers = [1, 2, 3, 4, 5];

var returnValue = arrayOfNumbers.find(value => value == 3);
console.log(returnValue);
```

This returns the number 3 because it exists in the array. If we search for a value that does not exist in the array it would return undefined.

The find method returns the first value that meets the function to return criteria. If in the above example we changed the criteria to 'value > 2' this would return the value 3 because it is the first value in the array that is greater than 2 even though 4 and 5 are also greater than 2.

```
var arrayOfNames = ["John", "Mara", "Peter", "Abigail"];

var returnValue = arrayOfNames.find(value => value == "Mara");
console.log(returnValue);
```

The find method also works with string data types.

The if statement can be used to check if the return value returns undefined in which case the value does not exist in the variable else the value does exist in the array (or vice versa using the logical not (!=) operator).

3.15 ForEach Loop in Arrays

ES6 introduces the For Each loop which allows you to look through each element in an array. The classic way of achieving this in ES5 was to use a for loop as seen below.

ES6 introduces the .forEach( ) method and every array has this method. This method takes in a callback function that run for each element in the array and is accessed via the value parameter. The second example code below achieves the same result as the for loop using the ES6 .forEach( ) method but written in a much simpler and cleaner syntax.

```
let names = ["John", "Alan", "Jess", "Dorothy"];

for(var i = 0; i < names.length; i++) {
    console.log(names[i]);
};
```

```
let names = ["John", "Alan", "Jess", "Dorothy"];

names.forEach(function(value) {
    console.log(value);
});
```

3.16 ES6 String Methods

The .repeat( ) method is a new method which repeat a string variable by a specified number of times. This new string can be stored in a new variable.

```
let myString = "Repeating String";

let repeatString = myString.repeat(5);
console.log(repeatString);          Repeating StringRepeating StringRepeating StringRepeating StringRepeating String
```

ES6 also introduces a new method to allow you to search for sub strings within a string. The .startsWith( ) method searches for a text that starts with whatever is passed as the method argument. If the substring exists then the method will return true else it would return false.

```
let myString = "Hello World";

let valueExist = myString.startsWith("H");
console.log(valueExist);          true
```

The .endsWith( ) method searches for a text that ends with whatever is passed as the method argument and returns true or false if there is or is not a match.

```
let myString = "Hello World";

let valueExist = myString.endsWith("ld");
console.log(valueExist);          true
```

The .includes( ) method will search inside the string variable for the substring (i.e. the argument passed into the variable) to return true or false if the substring exist or does not exist anywhere within the string.

```
let myString = "Hello World";

let valueExist = myString.includes("or");
console.log(valueExist);          true
```

3.17 Arrow Functions

ES6 introduces a new way to work with anonymous functions which is called the arrow function. The arrow function allows us to write a function in a much more simpler and readable syntax. Below is an example of a forEach function using a callback function using the ES5 syntax and the same function using the ES6 arrow function.

```
var nameArray = ["John", "Isabel", "Adam"];

nameArray.forEach(function(value) {
   console.log(value);
});
```

```
var nameArray = ["John", "Isabel", "Adam"];

nameArray.forEach(value => {
   console.log(value);
});
```

As we can see the syntax is much more readable. The arrow function also allows to remove the curly brackets if the code block has a single line of code.

```javascript
var nameArray = ["John", "Isabel", "Adam"];

nameArray.forEach(value => console.log(value));
```

Disclaimer: The arrow function will need the round brackets if there are more than one arguments to the function. The curly brackets is required if the function has more than one line of code for the code block.

It is called the arrow function because it uses the equal and greater than sign together (=>) to create a right pointing arrow.

3.18 Number Methods in JavaScript

The NaN (Not a Number) datatype is a special data type in JavaScript. This data type means that the value is not a number and cannot be converted into a number. To check if a variable value is a number we can call on the global Number object and then using the .isNan( ) method on the global object. As a parameter we would pass in the value we wish to check whether or not it is a number. If the value returns a NaN value then the .isNan( ) method will return true else it will return false.

```javascript
let num = 20;

console.log(Number.isNaN(num));
```

Disclaimer: The isNan( ) method would return false if we pass in a string or boolean because in JavaScript a string or boolean data type can be converted into a number.

In Javascript there is a special number called Infinity. Infinity in maths is not a number but a concept but in JavaScript Infinity is like a number object. We can have both a positive and negative Infinity value in JavaScript. Whenever a number is divided by 0 in JavaScript the value returned will always be Infinity. To check whether a number is of the type Infinity JavaScript has a .isFinite( ) method on the global number object. This is used in the same way as the isNan( ) method as seen above. The .isFinite( ) checks whether a number is a finite number and returns true. If the method returns false then the method indicates the value is infinity.

```javascript
console.log(Number.isFinite(100 / 0));
```

There are other methods on the global Number object such as .IsInteger( ) and .IsSafeIntefer( ) that work similar to the above to return true or false values if it meets the methods requirements i.e. whether the value is or is not an Integer value.

A number truncation is a process to cut out the decimal point of a number i.e. truncating a decimal number into a integer. The Math global object which has many common math methods such as floor, random, round, etc. that we can use. The .trunc( ) method will return the passed in argument as a truncated value.

```javascript
let num = 2.5;

console.log(Math.trunc(num));
```

Disclaimer: The trunc method does not round the number instead it returns the number without any decimal places. In the example on the left this will return 2.

ES6 introduces the .sign method on the Math global object to check whether a value is a positive or negative number. The method returns 1 when the number is positive and -1 when it is negative. When the value passed in as the argument is 0 i.e. neither positive nor negative the value returned is 0. If a value that is not a number is passed in as the argument the method will return NaN as the return value.

3.19 Date and Currency Formatting in JavaScript

Different countries in the world have different date formats for example in the UK the format is DD/MM/YYYY while in USA the date format is MM/DD/YYYY. ES6 introduces a new localisation library which is the Intl global object.

The .DateTimeFormat( ) method requires one parameter which is the country which we want to convert the date format to for example en-US is the American format while en-GB is the UK format. We use this method to create a converter object which we can then use to convert dates into the format specified using this converter object.

The .format( ) method returns a string representation of a date value and this method takes in a date object which we would create using the global new Date( ) constructor. This constructor takes in a date in the format of YYYY-MM-DD.

The converter object will convert the date into the specified format as seen in the example to the right.

```
let dateConverterUS = new Intl.DateTimeFormat("en-US");
let dateConverterUK = new Intl.DateTimeFormat("en-GB");

let dateToConvert = dateConverterUS.format(new Date("2020-06-28"));
console.log(dateToConvert);                                    6/28/2020

let dateToConvert2 = dateConverterUK.format(new Date("2020-06-28"));
console.log(dateToConvert2);                                  28/06/2020
```

The Intl library also introduces a way to format currencies in JavaScript. This works in a similar way to the Date formatting. We create a new variable and set its value to the Intl global object and then call on the .NumberFormat( ) method which takes in a parameter of the localisation as a string. The second argument is an options object where we specify the style and currency.

Disclaimer: Some countries in the world have more than one currency. Some countries use the comma (,) as the decimal separator.

We can then use this object to call on the .format( ) method passing in the integer/decimal number we wish to convert into a currency value. The syntax can be seen demonstrated in the below example.

```
let currencyUS = new Intl.NumberFormat("en-US", { style: "currency", currency: "USD" });
let currencyUK = new Intl.NumberFormat("en-GB", { style: "currency", currency: "GBP" });

let currencyToConvert = currencyUS.format(20.40);
console.log(currencyToConvert);                      $20.40

let currencyToConvert2 = currencyUK.format(30);
console.log(currencyToConvert2);                     £30.00
```

3.20 Array Includes, String Padding and Exponential Operation

The array .includes( ) method is a feature introduced in ES7 and returns true if a data exists inside an array and false if the data

```
var randomArray = ["John", 200, 173, "Jessica", 53.24, 100.42, "Diana"];

console.log(randomArray.includes(173));
```

does not exist inside of an array. We simply call the .includes( ) method on the array object and pass a argument of the value to find in the array.

ES8 introduces a new feature called string padding which adds spaces/padding at the beginning or the end of a string. The two new methods we can use on string variables are .padStart( ) and .padEnd( ) and these take in a integer as

```
let myString = "John";

console.log(myString.padStart(10));
console.log(myString.padEnd(10));
```

the first parameter/argument. This will ensure the string contains the number of characters specified in the argument and uses spaces as padding to ensure the number is met. Therefore, in the example, John is 4 character and the padding will add 6 spaces at the start/end of the string to make it 10 characters long.

Both methods can also take in a second argument of a string which will substitute the space padding with the string instead. In the second example, the padding will add 6 periods (.) at the start or end of the string instead of spaces.

```
let myString = "John";

console.log(myString.padStart(10, "."));
console.log(myString.padEnd(10, "."));
```

ES7 introduces the exponential operator which is two asterisk (**). A single asterisk syntax indicates to JavaScript to perform a multiplication while two asterisk indicates to JavaScript to perform a exponential operation. This is demonstrated in the example on the left where the first line of code does a multiplication of 3 x 3 = 6 and the second line of code does a exponential operation of 3 to the power of 3 i.e. 3 x 3 x 3 = 27.

```
console.log(3 * 2);    9
console.log(3 ** 3);   27
```

3.21 Object Values and Object Entries

ES8 introduces 2 new features of object values and entries. These methods show us the values of an object in an array format.

To use the object values feature we need to call the global Object property which has many methods to help us work with objects in JavaScript. The method we are interested in is the .values( ) method. This method takes in the object as an argument/parameter and will return an array of all the values of that object as seen in the below example.

```
let person = { name: "John Doe", age: 50, favouriteColour: "Blue" };

console.log(Object.values(person));                    [ 'John Doe', 50, 'Blue' ]
```

The object entries is a more robust feature because it shows both the name and value of each property. To use this feature we use the global Object and call on the .entries( ) method passing in the object as the argument/parameter. As seen in the example below this returns a multidimensional array i.e. an array containing sub-arrays for each key:value pairs i.e. properties.

```
let person = { name: "John Doe", age: 50, favouriteColour: "Blue" };      [
                                                                            [ 'name', 'John Doe' ],
                                                                            [ 'age', 50 ],
console.log(Object.entries(person));                                        [ 'favouriteColour', 'Blue' ]
                                                                          ]
```

This is a really powerful feature and we can use a for loop or a forEach loop to loop through the array and for example return only the key values which are all stored on index[0] of the sub-array. Examples of using either loops is provided below:

```
let person = { name: "John Doe", age: 50, favouriteColour: "Blue" };
let arrayOfEntries = Object.entries(person);

for(let x = 0; x < arrayOfEntries.length; x++) {          name
    console.log(arrayOfEntries[x][0]);                    age
};                                                        favouriteColour
```

```
let person = { name: "John Doe", age: 50, favouriteColour: "Blue" };
let arrayOfEntries = Object.entries(person);

arrayOfEntries.forEach(value => {                         name
    console.log(value[0])                                 age
});                                                       favouriteColour
```

These two new features are very powerful because it allows us to see what is within our objects in JavaScript.