Javascript

- This is a concise introduction to **Javascript (JS)** created by John Coumbe for Framework Training
- This document covers five major topics:

```
JS fundamentals
JS ES6
Functional programming
Asynchronous JS
Creating dynamic web pages with JS
```

JS fundamentals

- Working in Javascript requires an understanding of these four language fundamentals:

```
Variables
Arrays
Objects
Functions
```

Variables

- Consider a simple program which contains one light bulb. Clicking on the bulb turns it on or off. The program needs to keep track of the **state** of the light bulb.
- To track of state in JS we need **variables**.

```
Variables are temporary, named places in memory where we can store
values that will change during the lifetime of the program.
```

- This code creates a simple boolean variable and changes its value

```
let light = false;
light = true;
```

- We can toggle the value of a boolean variable using the **logical-not** ! operator.

```
// Toggle the value from true to false, or false to true
light = !light;
```

- Variables are **case-sensitive**. Lowercase, no-space names are a good choice.

```
let city = "Oxford";
console.log(City); // error: names are case sensitive.
```

- JS variables are **weakly-typed**. You can change any type of variable to any other type and JS does not throw an error.
- This relaxed approach to variable types can be a major source of bugs.

```
let score = 4;
score += "1"; // score now contains the string "41"
```

- *Advanced tip: consider using Typescript to spot type errors (www.typescriptlang.org) on larger projects.*

The LET and VAR keywords

- The examples above use the JS **ES6 let** keyword.
- Variables can be created with the older JS **ES5 var** keyword. But using var introduces several problems.
- Firstly, **block scope** does not work when var is used.

```
for ( var j=0; j <5; j++) {
    console.log(j);
}
// Variable j still exists here outside the for-loop
```

- Secondly, variables created with var are **hoisted**. They come into existence with a value of undefined at the start of their containing scope, not on the line where they are declared.
- Use let to avoid these problems. Variables created with let are **not hoisted and do have block scope**.

```
If you inherit a legacy project which uses keyword var, be careful
about changing the code to use let.
```

JS primitive and complex values

- One way to think about the JS language is to divide it into two groups.
- On one side are primitive values: **numbers, strings, boolean, null and undefined**.

- Primitive values are atomic: they contain a single value.
- On the other side are more complex object-like values: **objects, arrays, functions, classes**.
- Complex values have a structure and contain multiple values.

Primitives

- Here are 5 primitive variables. They each hold a single value.

```
let city = "Seville";
let year = 2020;
let smoker = false;
let project = null;
let village; // undefined.
```

- If we attempt to attach properties to them, JS fails silently.

```
city.nation = "Spain"
// FAILS : city.nation is undefined.
```

- **Null** means the developer has intentionally assigned an empty/nothing value to a variable.
- **Undefined** is the default if a variable has not been assigned any value.

Primitives: copy by value.

- If we a copy primitive, it is **copied by value**.
- The new variable is a separate independent variable.

```
let result = 4;
let score = result;
score++;
// result contains 4, score contains 5.
```

- We can compare the contents of two primitive values.

```
let city = "Paris";
let destination = "Paris";

console.log(city === destination); // true.
```

Complex values : copy by reference

- Objects and arrays behave differently.
- There are **copied by reference**. Simple assignment creates a new pointer to the same thing in memory.
- Variables "person" and "next" point at the same object.

- Variables "lottery" and "lotto" point at the same array.

```
let person = { name:"Fred", age:64 }
let next = person;
let lottery = [ 5,6,7,8,12,67 ];
let lotto = lottery;
```

- This behaviour is a major source of bugs.
- The ES6 **spread operator** (covered later) can be used to create true separate copies of an array or object.

Constants

- In JS ES6, variables declared with **const** are immutable.
- The variable cannot be re-assigned.

```
const PLAYER = "Ronaldo";
PLAYER = 42; // error
```

- Consequently, it makes no sense to attempt to create a const with no initial value. This will throw an error.

```
const VILLAGE; // error
```

- Note that for complex constants (arrays,objects) they cannot be re-assigned.
- But **their contents can be changed**.

```
const CITY = { name:"Seville", temp:40 }
CITY.temp++; // This property changes to 41.
```

- Many developers choose const as their **default choice** for a variable.
- Many variables do not need to change their value after initialisation.
- Choosing const avoids unexpected value changes which are a common sources of bugs. Immutable software is more reliable.
- Choose **let** if a variable needs to change value at runtime.

Strict equality

- The JS **loose equality operator ==** converts both sides of an expression to the same type and then makes a comparison.

```
let result = 4;
console.log(result == "4"); // true
```

- The **strict equality** operator only returns true if both sides of an expression are the same value and the same type.

```
console.log(result === 2+2); // true
```

Truthy/falsy

- In JS almost all expressions are equivalent to boolean true.
- All of these IF expressions will be treated as boolean true.

```
if( "abcd" ) {}
if( 4567 ) {}
if( [2,4,6,8] ) {}
if( {city:"Paris"} ) {}
```

- There is a short list of 6 values which are treated as boolean false.

```
false, undefined, null, 0, NaN, ""
```

Arrays

- Arrays hold a **related list of things** in a single variable.
- It is likely, but not essential, that all the elements of an array are of the **same type**.

```
let fruit = [ "apples","pears" ];
let temperatures = [ 24,26,22,28,40 ];
```

- The elements in an array are **zero-indexed**.
- The index runs from zero to length-of-the-array-minus-one.

```
let letters = [ "a","b","c","d","e" ];
letters[0]; // "a"
letters[4]; // "e"
```

- We can add and remove elements from the back of the array using **push** and **pop**.
- Here we create an empty array, add one element to the end, remove it, and then test the length of the array.

```
let cities = [];
cities.push("Seville")
let last = cities.pop();
console.log( cities.length );  // zero
```

- We can use **unshift** and **shift** to add and remove elements from the front of the array.

```
cities = [];
cities.unshift("Seville")
let first = cities.shift();
console.log( cities.length );  // zero
```

Tips for working with arrays

- Change arrays in simple ways.
- Avoid adding and removing elements from the **middle** of an array. It is easy to write buggy code that introduces undefined elements.
- Avoid writing code that refers to specific index numbers.
- You cannot make assumptions about the size of an array.

```
let songs = []
songs[17] = "Yesterday"

// songs now contains 17 nulls and one string.
```

Limitations

- Arrays do not contain metadata.
- An array works as an unordered collection of values.

```
let films = [ "Taxi Driver","North By North West" ]
```

- Problems can arise if we attach meaning to specific elements

```
let person = [ "John","Elton",42,"Green","Street" ]
```

- Does this describe "John Elton" from "42 Green Street"? Or "Elton John", aged 42, who votes "Green" and lives in the Somerset village of "Street"? If the first element contains 42, what do we assume about the other elements? If 42 is missing and there are only four elements, what do we do?
- Furthermore, when we consider a real world problem like a basket of shopping, a simple array is not up to the task.

```
let basket = [ "bread","yoghurt","milk","apples" ];
```

- What is the size/type/price/brand/stock-code for this bread?
- To remove this ambiguity we need to add metadata using **objects**.

Objects

- Objects are collections of **name-value pairs**.
- Here two name-value pairs describe the name and temperature of a city.

```
let city = { name:"Madrid", temp:28 };
```

- Objects work because their structure and meaning is clear.
- Objects are **mutable**: they can be modified later.

```
city.capital = true;
city.nation = "Spain";
```

- Every object is unique and can never be equal to another, even if their contents are the same.
- Objects are compared by identity.

```
let italy = { capital:"Rome" };
let italia = { capital:"Rome" };

console.log( italy === italia ); // false
```

- Using simple assignment, we create a copy by reference problem.
- Here both variables point to the same object and so the equality expression returns true.

```
let florence = { population:1.4 };
let firenze = florence;

console.log(florence === firenze); // true
```

Objects : JSON

- Javascript object notation (JSON) uses a **string representation** of objects.
- It has become a widely used **data format** for exchanging information between clients and servers.
- **JSON.stringify** converts a JS object to a string.
- **JSON.parse** converts a JSON string back into a JS object

```
let city = { name:"Madrid", temp:28 };
let s = JSON.stringify( city )
// "{"name":"Madrid","temp":28}"
city = JSON.parse( s );
```

- Local Storage is a browser-specific domain-specific API built into modern browsers for persisting non-secure data.
- We can use these JSON functions to store and retrieve data from localStorage.

```
let city = { name:"Madrid", temp:28 };

// Store
localStorage.city = JSON.stringify( city );

// Retrieve, checking for non-existence
city = localStorage.city ?
JSON.parse( localStorage.city ) : {} ;
```

Arrays of objects

- An array of objects combines the best features of arrays and objects to describe a **two-dimensional table of data** in one variable.

```
let weather = [
    { name:"Oslo", temp:-4 },
    { name:"Paris", temp:12 }
];

console.table( weather );
```

- We can iterate over an array of objects using the **forEach** method. It passes one object at a time to a function.

```
function describe( city ) {
    console.log( city.name, city.temp)
}

weather.forEach( describe );
```

Functions

- Functions are flexible, reusable pieces of code.
- In this example, calcArea is the **name** of the function.
- The **parameters/arguments** a and b allow us to pass different values into the function
- Curly braces {} mark the start and end of the **function body**.
- The **return statement** passes the calculated value back out of the function.
- The function defines a **locally scoped** variable area, which only exists temporarily at the moment that the function is running.

```
function calcArea(a,b) {
```

```
    let area = a+b;
    return area;
}
```

- A function does not do anything until it is **run, called, invoked or executued**.

```
let hall = calcArea(10,2);
let bathroom = calcArea(6,4);
```

- This example is a **pure** function. It does not change anything outside of the scope of the function.
- A pure function is predictable. Given the same parameters it will always return the same value.

Javascript ES6

- JS ES6 introduced major changes to the syntax of the language.
- This section covers these ES6 features:

```
Arrow functions
Destructuring
Template literals
The spread operator
Classes
Modules
```

Arrow functions

- JS ES6 introduced a new **concise syntax** for writing functions.
- Arrow functions also solve scope problems relating to keyword "this" *explained later in the section on classes.*

Arrow functions: one-line function

```
    const double = n => n*2;
```

- We define an arrow function using **const** or **let**.
- Keyword **function** is not required.
- The **name** of the function is double.
- We pass in one **argument** named n.
- The fat-arrow indicates where the **function body** begins.
- The function implicitly **returns** a value of "n*2"

```
double(2); // returns 4
```

Arrow functions: multiple arguments

- If we add pass in more than one argument, we need to wrap the arguments in parentheses ().

```
const add = (a,b) => a+b;
add(2,2);
```

Arrow functions: no arguments

- If we pass in no arguments, we need to include empty parentheses.

```
const nextYear = () => 2020 ;
nextYear();
```

Arrow functions: multiple-line functions

- Once the body of the function contains more than 1 line of code, two things change.
- We need to wrap the function body in **curly braces** {}.
- We also need to add back in an **explicit return** statement.

```
const addDebug = (a,b) => {
    console.log(a,b);
    return a+b;
}

addDebug(4,6);
```

Arrow functions: returning objects

- If an arrow function returns an object, we need to add **additional parentheses** ().
- This removes the ambiguity arising because {} are wrapped around an object, but also mark the start/end of a function body.

```
const makeCity = (name,temp) => ({ name,temp })
makeCity("Seville",40); // returns {name:"Seville",temp:40}
```

Destructuring

- We write a lot of **boilerplate code** to extract values from arrays and objects.

```
let first = songs.shift();
let last = songs.pop();

let age = person.age;
let name = person.name;
```

- Destructuring is a concise syntax for extracting multiple values from an array or object in one statement.
- Destructuring patterns can be confusing. Destructuring happens on **the left-hand side** of an assignment statement.

Object destructuring

- Object destructuring can create multiple variables in one statement.

```
const film = {title:"Jaws",director:"Spielberg"}
const { title,director } = film;
```

- Variables can be created and renamed in the same statement.

```
const { title:tl, director:dr } = film;
```

Array destructuring

- The elements of an array do not have names.
- You are free to assign any valid name to each element.

```
const person = ["Robert De Niro", "Taxi Driver"];
const [actor,film] = person;
```

Destructuring function return values

- Destructuring allows us to create functions which return multiple values, wrapped in an array or object.

```
const makeFilm = (actor,title) => ({ actor,title })
```

```
const {actor,film} = makeFilm("De Niro","Taxi Driver")
```

Destructuring function arguments

- We can destructure objects as they are passed into a function.
- This serves as self-documenting code and simplifies the function body.

```
const city = {name:"Seville", temp:40}

const weather = ({name,temp}) =>
`${name} is ${temp} degrees`

weather( city )
```

Destructuring with default values

- We can set up default values to handle the case where a destructured object may be incomplete.
- Here the "year" variable will default to 2020 if that property is missing from the object "trip".

```
let trip = { city:"Seville" };
let { year=2020, city } = trip;
```

Destructuring of nested objects

- Nested objects require additional syntax.

```
let flight = { type:"business" ,from: { airport: "Gatwick"}}
let {type, from:{airport}} = flight
```

- We can rename variables from nested objects.

```
let {type:typeTrip, from:{airport:flyFrom}} = flight
```

Template literals

- A **template literal** is a special type of string.
- It starts and ends with the special **back-tick** character
- It can be written over **multiple lines**.

- A template literal uses **string interpolation**. This means that variables and expressions wrapped in ${} syntax can appear within the literal.
- Here "markup" is a template literal containing two variables, city and temp. The result is a string which can inserted into a DOM element

```
const city = "Seville";
const temp = 40;
const markup = `<p>${city}<span>${temp}</span></p>` ;
const el = document.querySelector(".weather")
el.innerHTML = markup;
```

The spread operator

- Using simple assignment with arrays and objects introduces **copy by reference problems**.

```
let lotto = [2,4,6,8,10];
let lottery = lotto; // Both variables point to same array

let city = { name:"Rome" }
let capital = city; // Both variables point to same object
```

- The spread operator solves this problem. It splits an array or object into its individual parts.
- These parts can then be reassembled within a new empty array or object.

```
let lottery = [...lotto];
let capital = {...city};
```

- These are now separate variables which can be changed without affecting the originals.
- We can **modify/augment** the new variable at the time of its creation.

```
lottery = [ ...lotto, 12 };
capital = { ...city, year:2020 }
```

- Care needs to be taken when working with arrays of objects.
- This code creates a **new array containing the same old objects**.

```
let hols=[{city:"Oslo",temp:4},{city:"Paris",temp:2}];
let trip = [...hols];
```

- One solution to this problem is to map over the array applying the spread operator to each object.

```
let trip = hols.map(city => ({...city}))
```

Classes

- ES6 adds classes to Javascript allowing us to build projects using an **object-oriented** programming style.
- This code defines a class describing an animal.

```
class Animal{}
```

- We can create multiple object-instances from this class. Each object uses the same methods/functions but maintains its own data.
- This code **instantiates** two objects from the class.

```
const duck = new Animal();
const goat = new Animal();
```

Classes : constructors

- The constructor method runs automatically when an object is created.
- Parameters can be passed to the constructor.

```
class Animal{
    constructor(s) {
        this.sound = s;
    }
}
const duck = new Animal("quack");
```

- This code has created a class property called "sound" which is now in scope in any method of the class.

```
class Animal{
    constructor(s) {
        this.sound = s;
    }
    speak() {
        console.log(this.sound);
    }
}
```

```
const dog = new Animal("woof");
dog.speak();
const duck = new Animal("quack");
duck.speak();
```

Classes : arrow functions and scope

- In the example above, when we call **dog.speak()** the value of keyword "this" inside the speak method correctly points to the object "dog".
- **But keyword "this" does not always work in the same way.**
- Here method speak runs after a one second delay.
- The value of "this" inside speak depends on **how a function is called, not on where it is written**.

```
window.setTimeout( duck.speak , 1000 );
```

- The value of "this" inside speak will be set to the global namespace "window" or undefined.
- The complex scope rules surrounding "this" are a common source of bugs and confusion.
- **Arrow functions** can be used to solve this scope problem.
- An arrow function works out the value of "this" inside the speak method from the enclosing **lexical scope**. Here, it means that "this" now points to the dog instance of the Animal class.

```
speak = () => console.log( this.sound );
```

- *Classes are syntactic sugar. The ES6 class syntax is designed to be familiar for programmers from other object oriented languages. But under the hood, it still uses prototypes and prototype-based inheritance from JS ES5.*

Modules

- The ability to split code into **modules**, which define their own **scope**, helps to structure and organise an application.
- JS ES5 required you to build your own solution by creating module patterns using closure and immediately invoked function expressions (IIFEs).
- **ES6 modules** allow us to break code into separate files.
- By default everything inside that module is locally scoped.
- Variables only become visible from outside the file if we **export** them.
- Modules avoid polluting the global namespace by restricting the scope of variables.

```
// ======== utils.js
const digit = 4;
const quad = (n) => n*digit;
export {quad}

//  ======== code.js
import {quad} from "./utils.js";
quad(2);
```

Functional programming

- Functional programming uses methods like **sort, reduce, forEach, map and filter** to transform data in a concise way.
- These methods are **pure**. They do not change the original data.
- They can be **chained** together in a pipeline to acheive more complex transformations.
- Functional programming fits well with **ES6 arrow** functions.
- Sort, reduce, map and filter are **pure** functions. They

Functional programming: map

- Map applies a function to every element of an array. It returns a new array.

```
let sequence = [2,4,6,8,10];
sequence.map(n => n*2); // returns [4,8,12,16,20]
```

- Map always returns the **same number** of elements.
- But the nature of each element can differ from the original.

```
// An array of objects
let spain=[{city:"Madrid",temp:40},{city:"Vigo",temp:28}];
// Create an array of strings
let names = spain.map( city => city.name );
```

Functional programming: filter

- Filter returns a subset of the original array applying a boolean function to each element.

```
const italy = [{name:"Rome",temp:28},{name:"Pisa",temp:41}
// Filter in only cities over 40 degrees
const result = italy.filter(city => city.temp > 40)
```

Functional programming: reduce

- Reduce transforms an array of values into a single value.

```
const series = [ 8,2,6,4 ]

const smallest = (a,b) => Math.min(a,b)
series.reduce( smallest ); // returns 2

const add = (a,b) => a+b
series.reduce( add ); // returns 20
```

- Note: attempting to reduce an empty array will throw a run-time error.

```
const basket = [];
basket.reduce( add ); // Error
```

- We can handle this edge case by passing a second argument which is used as the return value if the array is empty.

```
basket.reduce( add, 0 );
```

Functional programming: sort

- Sort uses a compare-function which works on adjacent pairs in an array.
- The rules for how compare-fungtions work are quite complex. It is worth reviewing the Mozilla sort page
- This example sorts an array of objects in ascending temperature order.

```
const cities = [
{ name:"Madrid",temp:28}, { name:"Seville",temp:24} ]

cities.sort( (a,b) => a.temp - b.temp )
```

Functional programming: chaining

- More complex data transformations are possible by chaining together map, reduce, filter and sort.
- The order of the chain matters. In most cases, use filter before map.

```
// Create an array of the names of hot cities.
```

```
const cities = [
{ name:"Madrid",temp:28}, { name:"Seville",temp:40} ]

const hotCities = cities
.filter( city => city.temp >= 40)
.map( city => city.name )
```

Asynchronous programming.

- Some tasks are asynchronous (async) in nature.
- Reading data from an internet server takes an **unknown amount of time** and will have an **unknown outcome**.

Promises

- A promise is an object for managing **async** situations.
- A promise changes **state** once from pending to either fulfilled (succeeded) or rejected (failed)
- A **then** method runs if the promise is fulfilled.
- A **catch** method runs if the promise is rejected.

```
function openDoor() {
    return new Promise(
        function (resolve,reject) {
            let state = Math.random() > 0.5;
            state ? resolve("Open") : reject("Shut")
        }
    )
}

openDoor()
    .then( s => console.log(s))
    .catch(e => console.log(e))
```

- Promises allow us to write **flat non-nested code** to handle a variety of async cases.
- The return value of code inside a then method is itself a promise. This allows then methods to be **chained** together.

Fetch

- Fetch is a **Promised-based API** built into modern browsers for performing **AJAX** calls (XMLHTTPRequests)
- The first then method handles an HTTP response object.
- It extracts and returns data from it as a promise to a 2nd then method.

```
fetch("spain.json")
.then( response => response.json())
.then( weather => console.table( weather ))
```

The DOM

- The DOM is a **tree-like memory-based** structure which the browser constructs at run-time from the HTML on the current page.
- Your JS code can modify The DOM at run-time.
- It is important that the DOM exists before your code runs. Otherwise race-condition bugs may occur.
- The JS keyword **document** points at the DOM.
- The querySelector method uses **CSS selector** syntax to search the DOM for the first matching element.

```
// Find the first element with a class of "city"
let el = document.querySelector(".city");
```

- The querySelectorAll method returns an array of DOM elements.

```
// Find all elements with a class of "city"
let cities = document.querySelectorAll(".city");
```

- Combined with **template literal** syntax, we can inject new content into an empty section.

```
// <section class="capital"></section>

let city = {name:"Rome",temp:24};
let para = `<p>${city.name}<span>${city.temp}</span></p>`

let el = document.querySelector(".capital");
el.innerHTML = para;
```