Advanced JavaScript

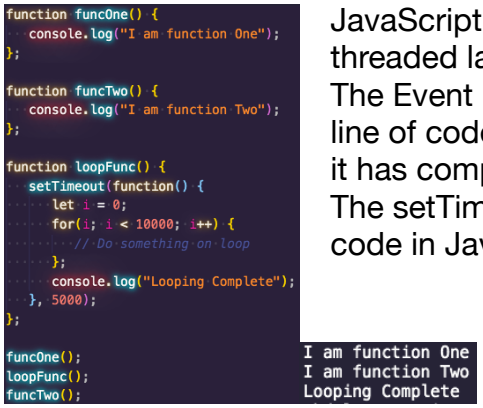
Section 3: Synchronous and Asynchronous Execution, Callbacks and Higher Order Functions & Strict Mode

---

1.1 Synchronous and Asynchronous Execution

JavaScript at its core is a synchronous language which mans that it executes code in sequence i.e. line by line. This therefore means the JavaScript code will not execute the next line of code (i.e. statement) until the first line of code has completed execution. For example if we had two functions of functionOne and functionTwo, the functionTwo cannot execute until functionOne has completely finished its execution.

The downside of synchronous code is that it can block execution of the next code until the current function is completely finished. Asynchronous code on the other hand allows multiple code to run at the same time asynchronously without having any issues of code blocking. This therefore allows different code to execute and complete at different times regardless the order the function was called.



JavaScript can have asynchronous execution but it is still a single threaded language and this is due to the JavaScript Event Loop. The Event Loop does not block execution and will allow the next line of code to execute and will only call the callback function once it has completed its execution and has returned something back. The setTimeout( ) API allows us to demonstrate asynchronous code in JavaScript.

In real world websites/webapps the code will make a request to the server to either fetch or write to a database. This server request generally takes some time to complete its execution and/or return something back. While this server request is occurring you would want to move onto the next line of code and do something else (i.e. you do not want users to sit around and wait for something to happen which would make for a very poor user experience). This is how a loader spinner is added when fetching data using the asynchronous code pattern.

1.2 Callbacks and Higher Order Functions

A callback is a function that is pass to another function. This is a design pattern when we want to write code asynchronously. Functions in JavaScript are “first class objects” which means that we can pass a function as an argument in another function and later execute it inside that function.

```
setTimeout(function() {
    console.log("I am a callback function");
}, 1000);
```

The setTimeout is an example of this whereby it takes in a function which it executes after a set amount of time in milliseconds. The anonymous function is what we would call a callback function.

```
setTimeout(myCallback, 1000);

function myCallback() {
    console.log("I am a callback function");
};
```

The setTimeout can also call-back a function we have declared and does not have to be an anonymous function. The first argument would be a reference to the function as an object.

The setTimeout function takes in two arguments the first is the function to callback whether that is an anonymous function or a reference to a function object and the second is the time the setTimeout should wait before it executes the function.

The .addEventListener is another example of a method that uses a callback function. The callback function is the second argument that gets called whenever the first argument event is triggered e.g. on a click event. Again, the .addEventListener can take either an anonymous function or a function object reference as the second parameter.

```
document.getElementById("button").addEventListener("click", function() {
    console.log("Button clicked");
});
```

```
function buttonClicked() {
    console.log("Button clicked");
};

document.getElementById("button").addEventListener("click", buttonClicked);
```

We are basically passing around a function and it looks like just a variable even though we know it is a function which is the uniqueness about JavaScript i.e. the function are specialised objects which allows us to pass them around like a variable.

It is important to note that we do not invoke the function using the open and closed round brackets after the function name because we do not want to immediately invoke the function because we would only want to callback the function at a later time i.e. on a event or after a set timer.

A Higher Order Function is a function that takes another function as an argument or returns a function. The array object has a special method called forEach. This method takes in a callback function as its argument.

```
const animals = ["Cat", "Dog", "Fish", "Zebra"];

animals.forEach(function(animal) {
    console.log(animal);
});
```

The .forEach( ) method loops through each array items which we can get access to and do something with it. In the example on the left we console log each animal in the list.

The .forEach( ) is a Higher Order function because it requires to take in another function as its argument. The forEach method will throw a TypeError if a function is not passed in as an argument.

```
const animals = ["Cat", "Dog", "Fish", "Zebra"];

animals.forEach(printAnimal);

function printAnimal(animal) {
    console.log(animal);
};
```

We can also pass in a declared function into the .forEach( ) method and this will continue to work. This is demonstrated in the example on the left which achieves the same output as the anonymous function example above.

Again we do not invoke the function but pass it as a reference function object. The function will get called for each item in the array.

There are other array methods which are Higher Order Functions i.e. take in a function as its argument. For example: the .filter( ) method takes in a function as its argument. This allows us to return a subset from the array. The .map( ) method also takes in a function as its argument and this allows us to return a new array from the original array.

```
const animals = ["Cat", "Dog", "Fish", "Zebra"];

console.log(animals.filter(function(someAnimal) {
    return someAnimal.startsWith("F");
}));
```

The .startWith( ) method is case sensitive and will return any items that matches whatever is passed in as the argument. In the example on the left this will return Fish only.

```
const animalsStartWithF = animals.filter(function(someAnimal) {
    return someAnimal.startsWith("F");
});

console.log(animalsStartWithF);
```

Note because the .filter( ) method returns a value we can assign/store this return value in a variable.

```
const people = [
    { name: "John Doe", age: 40 },
    { name: "Sue Finch", age: 55 },
    { name: "Chloe Dalton", age: 30 }
];

const peopleNames = people.map(mapPersonName);

function mapPersonName (person) {
    return person.name;
};

console.log(peopleNames);
```

```
[ 'John Doe', 'Sue Finch', 'Chloe Dalton' ]
```

In the example on the left the .map( ) function takes in a function as its argument. This can either be an anonymous function or a reference function object. This is also true for the .filter( ) example.

When using a reference function object as the argument we must not invoke the function right away for these Higher Order Functions.

The .map( ) method example returns back a new array object with only the first names extracted.

The .map( ) method is useful if we want to alter/transform the data as we extract it from the array to create a new array object (we do not have transform the data but we have the option available) while the .filter( ) method is useful for returning whatever we are looking for within the array without any alteration/transformation to the data extracted.

A Higher Order Function can also return a function inside of a function.

```
function numberStringify(num) {
    return num.toString();
};

const numToString = numberStringify(33);
console.log(numToString);
```

In the example on the left we can transform a number passed into the function into a string. The .toString( ) is a built-in method that we can call in JavaScript which reruns a string representation of the data type e.g. a number type into a string type.

```
function removeFirstLetter(str) {
    return str.substring(1);
};

const greet = "HEllo"
const fixedString = removeFirstLetter(greet);
console.log(fixedString);
```

Another example of a Higher Order Function which returns a function. The .substring( ) method allows us to return a substring starting from an index. In the example on the left this will return the string starting at index 1 which is the letter E and returns all characters after this index.

The concept of Callbacks and Higher Order Functions are simple; however, they can be become used for more complex executions. We can use both Callbacks and Higher Order Functions together.

In the example on the right we have a logMyNumber which takes in a function as its argument. At the end of the function we can invoke the callback function. Note: we could have named this argument anything like cb, something, etc.

We have another function called logger which console logs. We can therefore invoke the logMyNumber function and pass the logger function as its argument. The logger function will be called back once the loop completes execution.

Callbacks makes our code more dynamic/flexible i.e. we can create another function that logs something different. So long as we pass to the logMyFunction a callback function as a reference function object correctly, it does not matter which function we pass as the callback function. This therefore makes our function more flexible as seen in the example code to the right.

```
function logMyNumber(callback) {
    for (let i = 0; i < 1000; i++) {
        // Loop
    };
    callback();
};

function logger() {
    console.log("Loop Complete!");
};

function somethingDifferent() {
    console.log("I'm Done Looping!");
};

logMyNumber(logger);
logMyNumber(somethingDifferent);
```

1.3 Strict Mode

Strict mode was introduced in ES5 which means the feature is supported in all modern browsers. Strict mode places the code in a ‘strict’ operating context. This prevents certain actions and JavaScript will throw more exceptions to let you know that that your code is doing something that you should not do in your JavaScript code (i.e. prevent your code from failing silently less often).

```
function someFunction() {
    x = 10;
};

someFunction();
console.log(x);
```

In the example on the left, we would expect the code not to work because variables inside of a function is scoped to the function. However, we should notice that in the example we did not declare the variable (i.e. using the var, let, const keywords). What happens when we do this is that the variable x is not actually scoped to the function.

```
function someFunction() {
    x = 10;
};

someFunction();
x = 60;
console.log(x);
```

If we do not use the var, let or const keyword i.e. do not assign the variable; the variable will end up bubbling up the scope until it finds a declaration and if it does not find one it will attach itself to the global object i.e. the windows object. Therefore, we will have access to the variable x in the global scope and will be able to access the x variable outside the function. This also means we can reassign the variable later on which is not ideal as it can cause side effects silently.

Strict mode will throw an exception in this scenario and prevent us from doing the above. Strict mode can be used either in the global scope or to individual functions. To enable the strict mode we simply add the string “use strict”; in our code either at the very first line for global or in the function body for the function only.

```
function someFunction() {
    "use strict";
    x = 10;
};

someFunction();
x = 60;
console.log(x);
```

```
"use strict";

function someFunction() {
    x = 10;
};

someFunction();
x = 60;
console.log(x);
```

Strict mode will come in handy to prevent use/assign a variable without declaring it with the var/let/const keyword. This will throw a ReferenceError in the console.

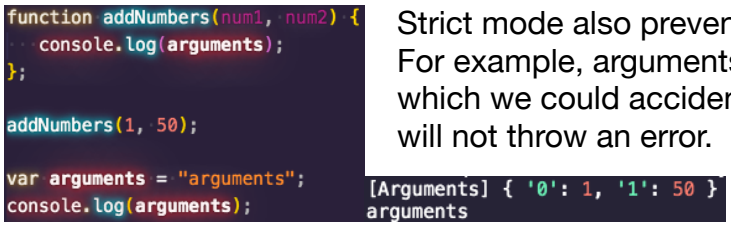
In JavaScript we can also delete variables or objects using the delete keyword to remove a variable/object from the global scope. This is not really good practice but it does exists in JavaScript i.e. we can delete variables/objects without actually removing the line of code which creates it. This can cause problems especially if working in a large team whereby the variable may be used in someone else function.

```
someVar = 123;
someObj = { name: "John Dow" };
console.log(someVar);    // Returns 123
console.log(someObj);    // Returns "John Doe"

delete someVar;
delete someObj;
console.log(someVar);    // Returns ReferenceError
console.log(someObj);    // Returns ReferenceError
```

Therefore, to save ourselves from this issue we can use the strict mode to prevent deletion of variables/objects/functions whatsoever. This will throw a SyntaxError in the console.

JavaScript allows you to use duplicate function parameter names without throwing an error. In the example on the right the addNumbers would return 4 rather than the expected 3 because num1 has been reassigned the second argument parameter. Strict mode will prevent this from happening and throw a SyntaxError in the console.

```
function addNumbers(num1, num1) {
    console.log(num1 + num1);
};

addNumbers(1, 2);
```



Strict mode also prevents us from redefining reserved words. For example, arguments is a reserved keyword in JavaScript which we could accidentally reassign its value and JavaScript will not throw an error.

This can have significant impact on our JavaScript code. Strict mode will prevent reassigning values of reserved keywords and will throw a SyntaxError in the console.

It is highly encourage-able to use Strict mode wherever possible in your code so that JavaScript can throw these errors wherever we try to do something we should not be doing in JavaScript without having any silent errors. This makes it much more easier to debug our code with these types of bugs/errors.

JavaScript was intended to be a very easy language to use; however, it can have a very annoying tendency to fail silently or kind of does not fail but does nothing at the same time i.e. there is no hint as to what is going wrong with the code to debug the issue. Therefore, Strict mode is highly recommended to use in your code.

There are other uses of Strict mode but the above examples are the common examples of what Strict mode is all about.

END