Advanced JavaScript

Section 1: DOM Manipulation, Objects and Functions

1.1 JavaScript Execution Environment

JavaScript is a interpreted language which means that it is run on the fly line-by-line instead of needing to be compiled (compare this to Java which is a compiled language i.e. before any code can be run it must be run through a compiler).

All code we write as developers has to be understood by a computer (i.e. in JavaScript this is the browser). This is the job of the JavaScript engine and every browser has a JavaScript engine of some sort. The job of the engine is to interpret the code the developer writes and tell the browser how to execute the code.

Example of JavaScript engines are: V8 (Chrome), Spidermonkey (FireFox), JavaScriptCore (Safari). As we can see different browsers have different engines and this means that sometimes browsers will behave differently.

When the engine starts to read the JavaScript code the following things occur:

1. Before any code is executed the global execution context is created. This is the global environment (i.e. the window in a web browser) - this is sort of the top level
2. Any time a function is executed, a new execution context gets created and gets added to the call stack

JavaScript is a single threaded language i.e. it can do one thing at a time (as opposed to a multi-threaded language which can do multiple things at a time). The call stack is a data structure that contains information on the order of function calls. The last function that gets called is the first function to get out of the call stack (i.e. LIFO).

1.2 DOM Manipulation

The DOM stands for Document Object Model and is an object-oriented representation of a webpage which can be modified with a scripting language such as JavaScript (definition from Mozilla Developer Network (MDN)).

HTML and JavaScript are two different languages and need a way to interact with each other. The DOM is neither HTML or JavaScript but rather an interface for JavaScript, HTML and CSS to interact with each other. We can think of the DOM as an API (interface) for interaction between the different languages.

In the example below we are using JavaScript to manipulate the DOM to interact with the HTML webpage. The document is an object with many different properties and methods we can use to manipulate the DOM. This does not mean that we are directly

manipulating/changing the HTML i.e. whatever we do with JavaScript might interact with the DOM but it does not directly interact with the HTML or CSS.
The createElement is a method we can call on the document object to create a new HTML element on the DOM such as a element. The innerText property allows us to add text to the element. This creates a new element but nothing is done with this element to add to the DOM/HTML.

The getElementByID method allows us to grab an element in the HTML and make it into an object representation of that element which is stored in a variable. We can use both elements which are stored in variables to manipulate the DOM. The .appendChild method allows us to append an element to another element.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Manipulation</title>
  </head>
  <body>
    <div id="hello">Hello</div>

    <script src="script.js"></script>
  </body>
</html>
```

```html
<script>
  var newElement = document.createElement("div");
  newElement.innerText = "World";

  var helloDiv = document.getElementById("hello");

  helloDiv.appendChild(newElement);
</script>
```

Important Note: We could have used the .innerText property to change the div with the id of hello with a different text. This does not change the HTML document; however, this does manipulate the DOM which is the HTML page representation and what gets rendered in the browser window.

With JavaScript we can manipulate a webpage via the DOM and its core this is what JavaScript code does with a browser and how we can make webpages interactive. We can use the document object model (DOM) to create elements, edit elements and remove elements from the webpage.

Whenever we manipulate the DOM we need for it to be ready first i.e. we need the HTML and CSS to be loaded into the browser and ready for us to manipulate. If we run our JavaScript to manipulate the DOM before it is loaded it might cause errors or the page not to function correctly. The JavaScript code is typically loaded into the HTML via <script> tags at the bottom of the body because the browser reads the HTML document from top to bottom which will load all the content of the webpage before loading the JavaScript which manipulates the DOM i.e. the JavaScript will only run after the DOM is completed loaded.

There are many ways to grab an element from a HTML document using the document object model for example we can use the getElementById, getElementByClass, querySelector and querySelectorAll methods. There are many ways to achieve the same results which is true in every programming language. Below is an example of JavaScript code selecting elements on a HTML document using the DOM object.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Manipulation</title>
  </head>
  <body>
    <div id="example">
      <p>Paragraph One</p>
      <p>Paragraph Two</p>
      <div id="inner-div">
        I am an inner div
      </div>
    </div>

    <script src="script.js"></script>
  </body>
</html>
```

```html
<script>
  const paragraphOne = document.querySelector("p");
  paragraphOne.style.color = "orange";

  const exampleDiv = document.getElementById("example");
  const paragraphOne = exampleDiv.querySelector("p");
  paragraphOne.style.color = "teal";

  const paragraphOne = document.querySelector("div p");
  const paragraphOne = document.querySelector("div > p");
  const paragraphOne = document.querySelector("div#example p");
  paragraphOne.style.fontSize = "50px";

  const allParagraphs = document.querySelectorAll("div#example p");
  allParagraphs[0].style.fontFamily = "courier";
  allParagraphs[1].innerText = "Paragraph 2 is cool";
</script>
```

The addEventListener is a special method which tells JavaScript to listen to an event on an element and when that event occurs to trigger the function. The method has a pointer reference to the function which is passed in as the second argument/parameter to the addEventListener method. In the below example, when the <button> element with an id of example-button is clicked it will add the showAlert function to the call stack and when it resolves it will alert the user with an alert message.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>DOM Manipulation</title>
  </head>
  <body>
    <div id="example">
      <button id="example-button">Click Me</button>
    </div>

    <script src="script.js"></script>
  </body>
</html>
```

```html
<script>
  const button = document.getElementById("example-button");
  button.addEventListener("click", showAlert);

  function showAlert() {
    window.alert("Thanks for clicking :D");
  };
</script>
```

This is a few example of how we can use JavaScript with the DOM to manipulate the HTML and CSS of webpages.

1.3 Objects

An object is a collection of related data and/or functionality which usually consists of several variables and functions (which are called properties and methods when they are inside objects). Therefore, we can consider objects as a data structure i.e. objects hold information to either store data and functionality.

We can think of objects like the real world objects i.e. an object in the real world have properties and functionalities e.g. all animals have properties/qualities such as name, gender, age, animal type, etc. as well as functionality such as walking, running, eating, noise it makes, etc.

To create an object we first need to create a variable and then assign its' value to a curly opening and closing bracket which denotes to an object in JavaScript. Inside the curly brackets are the properties and methods the object will hold. Each property have a key = value pair and are separated with comma's.

```js
const dog = {
  name: "Spot",
  age: 10,
};

console.log(dog.name);
console.log(dog["name"]);
```

To access a property we would call the object followed by a period and then the property name. This is known as the dot notation. The alternative is to use square brackets instead of the period and wrap the property name in quotations. This alternative is known as the bracket notation.

JavaScript objects can hold any data types for its properties whether it is a string, number, boolean, function, array, etc. To call a object method we would have to use the dot notation.

```js
const dog = {
  name: "Spot",
  age: 10,
  bark: function() {
    console.log("Woof");
  },
  speak: function(nameToSpeak) {
    console.log("Woof " + nameToSpeak + "!");
  }
};

dog.bark();
dog.speak("Bob");
```

The property name has the opening and closing brackets at the end to denote that we are calling an object method and not a property. We can also pass arguments/parameters to functions.

```
const cat = {
    name: ["Mister", "Kitten"]
};

console.log(cat.name[0]);
console.log(cat["name"][1]);
```

To access an array within an object is to call on the property followed by the index. JavaScript uses zero indexing notation. The index of 0 will return the first item in the array, in the example on the left this will return the name Mister.

```
const cat = {
    name: {
        first: "Mister",
        last: "Kitten"
    }
};

console.log(cat.name.first);
console.log(cat["name"]["last"]);
```

We can also have an object inside of an object. The dot notation or bracket notation can be used to grab the properties using a chain.

```
const cat = {
    name: {
        first: "Mister",
        last: "Kitten"
    }
};

cat.name.first = "Snow";
cat.favouriteFood = "Tuna";
```

We can reassign a property value by accessing the property and then using the assign operator (=) set it a new value as well as adding a new property to the object. Both are demonstrated on the example to the left.

We can add/remove things to/from an object and so it is definitely quite possible to use JavaScript in a really object oriented way.

The document is an object which we can use to communicate with the DOM.

The this keyword inside of an object refers to the object the code is inside. In the example below the this keyword is inside of the cat object. If we want the cat object to have access to it's own variables (properties) we would need to use the this keyword. Therefore, the this keyword in the context of using it inside of an object refers to the object itself.

```
const cat = {
    name: {
        first: "Mister",
        last: "Kitten"
    },
    favouriteFood: "Tuna",
    sayFavourite: function() {
        console.log("Meow fav food is " + this.favouriteFood);
    }
};

cat.sayFavourite();
```

The this keyword outside an object refers to the global window object. The window object is a DOM representation of the window the browser is in. Therefore, we should be very careful in what context the this keyword is being used in i.e. what object it is inside of.

1.4 Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure i.e. a set of statements that perform a task or calculates a value.

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

sayHello();
```

A function can be seen as a special type of object and what this means is that they are objects like any other object in JavaScript except they are different in that they can be called (i.e. we can invoke the function at a particular line by its name followed by open and closed round brackets).

Functions allow us to do one or a few small things which lets the developer write less code and prevents unnecessary duplication in code.

To create a function we use the reserved function keyword followed by the name of the function followed by an open and closed round brackets. Enclosed within the round brackets we can add parameters/arguments which can be used in the function. The code block lives inside of the curly brackets.

```
function addition(num1, num2) {
    console.log(num1 + num2);
};

addition(1, 2);
```

JavaScript does not care about data type and therefore in the example above for num1 and num2 we could pass in any values other than number and this will continue to work for example we can pass in two strings.

It is very common in JavaScript for something to return from a function. To return a value from a function we would use the reserved return keyword. The return keyword tells JavaScript to stop whatever it is doing and return on this line. If the return is given a value it will return that value to wherever the function was invoked. Anything after the return keyword in the function will not be executed. We can assign this value to a variable, console log the return value or do nothing with it.

```
function addition(num1, num2) {
    return num1 + num2;
};


const newNumber = addition(1, 2);
console.log(newNumber);
```

Functions allows us to reuse the code as many times as we want without having to re-write the logic over and over again following the principal of DRY (don't repeat yourself).

A function expression is similar to a function but has a slightly different syntax. We would first declare a variable and assign its value to a function expression as seen in the example on the right.

```
const addNumbers = function(num1, num2) {
    console.log(num1 + num2);
};

addNumbers(1, 2);
```

There are no real differences between the two ways in declaring functions but the one thing to be aware of is that the JavaScript engine does treat the two ways of functions slightly differently. If we try to invoke the function expression before we declare it this will throw a ReferenceError. However, the function declaration does not throw and error because JavaScript does a thing called hoisting (i.e. it puts function declarations at the top on the first read through)

```
addNumbers(1, 2);    //Throws error
const addNumbers = function(num1, num2) {
    console.log(num1 + num2);
};

addNumbers(1, 2);    //Hoisting No errors
function addNumbers(num1, num2) {
    console.log(num1 + num2);
};
```

If we are passing a function into another function or we want to call an anonymous function we would want to use a function expression.

An anonymous function is a function that has no name. The function expression examples above use anonymous function i.e. there is no name after the function keyword (note: we could have added a function name to the function expression after the function keyword). Function declarations must have a function name after the function keyword and cannot be anonymous.

```
button.addEventListener("click", function() {
    window.alert("I am an anonymous function!");
});
```

In the example on the right, when we click on a HTML button this will trigger the anonymous function to display an alert to the browser's window.

END