JAVASCRIPT

CSE5006 – LAB 3

TABLE OF CONTENTS

1. WHAT IS JAVASCRIPT?

JavaScript is a programming language with the original purpose of adding dynamic behaviour to web pages. Although JavaScript bears some superficial resemblances to Java, it is a completely different language, so don't let the name fool you. Over the years, multiple different revisions of the JavaScript specification have been released. It's possible to achieve maximum compatibility in the meantime by transpiling the code into an earlier version of the language.

So why JavaScript instead of Java or C++? Well, unfortunately, if you want to do client-side (in-browser) web development, you simply don't have much of a choice. JavaScript is the only programming language with native support in all current major web browsers (Safari, Chrome, Firefox, and Internet Explorer). To make these labs run more smoothly, we will be using JavaScript not only for client-side code but for the server-side part as well. Note that there are many more choices for server-side programming languages - for example, you could do that part in Java or C++.

2. BASIC EXPRESSIONS AND VARIABLES

Start the VirtualBox VM for this subject as you did in the first lab (you will need to start every lab by doing this). Once you have logged in, use Git to clone your simple website fork of from lab 1. Navigate to the project directory and open index.html in the web browser Firefox. Once in the web browser, press Ctrl+Shift+K to open the JavaScript console.

Enter some basic mathematical expressions into the console and observe the results printed out.

```
// INPUT
439 - 19
// OUTPUT
420

// INPUT
Math.PI * 10
// OUTPUT
31.41592653589793
```

As you can see, basic expressions look pretty much the same as they do in Java. However, variables are a bit different because in JavaScript, you do not specify the type of variables.

```
// INPUT
let var1 = 27;
// OUTPUT
undefined

// INPUT
var1
// OUTPUT
27

// INPUT
let var2 = 'Twenty-seven';
// OUTPUT
undefined

// INPUT
var2
// OUTPUT
'Twenty-seven'
```

Although you can use the same variable to store differently typed values, I don't recommend it in most cases as it can cause a lot of confusion. Note that in the example above, we used the **let** keyword to declare a variable. Alternatively, we can declare a constant (a variable that can't be reassigned) using the **const** keyword.

```
// INPUT
const y = 29.5;
undefined

// INPUT
y
// OUTPUT
29.5

// INPUT
y = 32.0;
// OUTPUT
// TypeError: invalid assignment to const `y'

// INPUT
y
// OUTPUT
29.5
```

In general, I prefer using **const** over **let** whenever possible. This convention makes it a lot clearer when a variable will be reassigned a new value later on.

2.1. EXERCISE 1

Try to declare a variable using **let** or **const**, and then declare it again. What happens? Put your answer to this exercise (and others) in a text file for your demonstrator to check later.

3. ARRAYS AND OBJECTS

3.1. ARRAYS

Arrays in JavaScript can be created using square brackets and may contain elements of any type. You can even have different element types within the same array, although this is usually discouraged.

```
// INPUT
const planets = ['venus', 'earth'];
// OUTPUT
undefined

// INPUT
planets
// OUTPUT
[ 'venus', 'earth' ]
```

There are a few built-in functions and properties available for JavaScript arrays, which you can read about in the JavaScript Array Reference.

```
// INPUT
planets.unshift('mercury'); // Add element to start
// OUTPUT
3

// INPUT
planets.push('mars'); // Add element to end
// OUTPUT
4

// INPUT
planets
// OUTPUT
[ 'mercury', 'venus', 'earth', 'mars' ]

// INPUT
planets [2] // Get 3rd element (indexing is zero-based)
// OUTPUT
'earth'

// INPUT
planets.length // Get number of elements
// OUTPUT
4
```

Note that we are able to change the array, even though it is a constant. This is because JavaScript only enforces that the reference is constant. As long as we don't reassign the value using "=", other types of mutation are fair game.

3.1.1. EXERCISE 2

Write code that takes the array [’c’, ’b’, ’a’] and reverses it, then concatenates it with the array [1, 2, 3]. Finally, join the elements in the result to produce the string “abc123”. Hint: use the JavaScript Array Reference mentioned earlier!

3.2. PLAIN OBJECTS

Objects in JavaScript are flexible ways to combine data, similar to structs in C/C++ or objects in Java. An interesting difference is that in JavaScript, we are not required to create a "blueprint" (in Java, a class) from which the objects are created. This means that we can use JavaScript objects as hash maps (dictionaries) as well.

```
// INPUT
const person = {name: 'Daniel', age: 20};
// OUTPUT
undefined

// INPUT
person.name // Access an attribute
// OUTPUT
'Daniel'

// INPUT
person.job = 'Painter'; // Add a new attribute
// OUTPUT
'Painter'

// INPUT
person
// OUTPUT
{ name: "Daniel", age: 20, job: "Painter" }
```

In ES6, there is a convenient shorthand that can be used when using object initializers with properties that match variable names.

```
// INPUT
const paintColour = 'red';
// OUTPUT
undefined

// INPUT
const car = { paintColour }; // Same as {paintColour: paintColour}
// OUTPUT
undefined

// INPUT
car
// OUTPUT
{ paintColour: "red" }
```

3.3. THE DOCUMENT OBJECT

When using JavaScript in the web browser, we have access to a special global object called **document**. This object contains a representation of the web page called the Document Object Model (DOM), which is constructed from the HTML source code. Using JavaScript, it is possible to manipulate the DOM via the **document** object, which will alter the web page. Please note that this is different from changing the HTML itself, as it only affects the instance of the page loaded in the browser. In other words, manipulating the DOM using JavaScript will not change the contents of **index.html**.

For example, let's say we wanted to replace the textarea with a greeting message using JavaScript. We can accomplish this by obtaining the div element with the ID "yellow box" and modifying its innerHTML property.

```
document.getElementById('cool-section').innerHTML = "Hello there!"
```

You should notice the page change after running the code. However, if you refresh the page, the DOM will be reset to its original structure.

Let's look at another example, this time setting the style attribute of the body tag to change the page's background colour.

```
document.body.setAttribute('style', 'background-color: green;');
```

JavaScript is extensively used to manipulate the DOM on most websites these days, enabling dynamic content and rich applications. In fact, some developers go as far as writing entire games in JavaScript that can be played directly in the web browser! Virtually anything that could have been accomplished with HTML alone can be achieved using JavaScript in conjunction with the document object. The advantage of JavaScript lies in its ability to facilitate changes to the page over time and respond to user input.

4. FUNCTIONS, METHODS, AND CLOSURES

4.1. TRADITIONAL JAVASCRIPT FUNCTIONS

The traditional way of creating functions is to use the **function** keyword.

```
// INPUT
function doubleValue(x) {
    return x * 2;
}
// OUTPUT
undefined

// INPUT
doubleValue(7)
// OUTPUT
14
```

Functions created in this way possess an interesting property that makes them well-suited for creating object methods: they automatically bind the **this** keyword to the object to which the method is currently attached. Let's examine an example.

```
// INPUT
const areaOfSquare = function() {
    return this.size * this.size;
};
// OUTPUT
undefined

// INPUT
const square1 = {size: 7, area: areaOfSquare};
// OUTPUT
undefined

// INPUT
const square2 = {size: 4, area: areaOfSquare};
// OUTPUT
undefined

// INPUT
square1.area()
// OUTPUT
49

// INPUT
square2.area()
// OUTPUT
16
```

There are a few new things happening in the above example. Firstly, we can observe that functions can be created without a name and assigned to variables, just like any other value. In this case, **areaOfSquare** will be a constant, meaning that, like any constant, it cannot be assigned a new value. Secondly, object methods can be created by assigning a function to a property. Lastly, the **this** keyword serves as a placeholder for the object to which the method is attached.

But what happens if we call **areaOfSquare** directly?

```
// INPUT
areaOfSquare()
// OUTPUT
NaN
```

That's because **areaOfSquare** is utilizing the "global this", which does not have a property called **size** defined. In JavaScript, when multiplying undefined by undefined, the result is **NaN** (not a number).

```
// INPUT
this.size
// OUTPUT
undefined

// INPUT
this.size * this.size
// OUTPUT
NaN
```

But what if we want to explicitly set **this** without attaching the function to an object? One way to achieve that behaviour is with **bind**.

```
// INPUT
const square3 = {size: 2};
// OUTPUT
undefined

// INPUT
const areaOfSquare3 = areaOfSquare.bind(square3);
// OUTPUT
undefined

// INPUT
areaOfSquare3()
// OUTPUT
4
```

As you can see, the value of **this** has been set to **square3** in the function **areaOfSquare3**.

4.1.1. EXERCISE 3

Create an object with a property **name** and a method **introduce**. When **introduce** is called it should return the string “Hello, my name is <name>”, where <name> is the string stored in the object’s **name** property.

4.2. ARROW FUNCTIONS

ES6 introduced a new style of functions commonly referred to as "arrow functions." Unlike the old style, arrow functions do not change the value of **this** depending on context. The value of **this** will always be whatever the value of **this** was when the function was defined.

```
// INPUT
const arrowAreaOfSquare = () => {
    return this.size * this.size;
};
// OUTPUT
undefined

// INPUT
const square4 = {size: 6, area: arrowAreaOfSquare};
// OUTPUT
undefined

// INPUT
square4.area()
// OUTPUT
NaN
```

Arrow functions are equivalent in meaning to using bind at the point of creation, like so:

```
const boundAreaOfSquare = function() {
    return this.size * this.size;
}.bind(this);
```

The code above makes it explicitly obvious that the **this** variable has been permanently bound to the "global **this**", and therefore, it will not be able to work like a method when attached to an object property.

So why use arrow functions? For one thing, they eliminate confusion about what the value of **this** is—it always stays the same.

However, the real benefit to arrow functions is how concise they can make code. For example, arrow functions have a special shorthand where the curly braces and return keyword can be omitted for one-line functions.

```
const addTogether = (a, b) => a + b;
```

Compare that to the old way of doing the same thing.

```
const addTogetherOld = function(a, b) {
    return a + b;
};
```

No matter which syntax you use, the functions are called in exactly the same way.

```
// INPUT
addTogether(5, 4)
// OUTPUT
9

// INPUT
addTogetherOld(5, 4)
// OUTPUT
9
```

We will be using arrow functions extensively throughout the labs in this subject.

4.3. CLOSURES

Both types of functions in JavaScript are able to "close over" variables in higher scopes.

```
// INPUT
const higherScopeConst = 42;
// OUTPUT
undefined

// INPUT
const doSomething = (x) => x + higherScopeConst;
// OUTPUT
undefined

// INPUT
doSomething(8)
// OUTPUT
50
```

No matter where we call **doSomething** from, it will always be able to “see” the value of **higherScopeConst**. You can think of it as if **higherScopeConst** got “bundled up” with the function.

5. HIGHER ORDER FUNCTIONS

A higher-order function is a function that takes another function as an argument and/or returns a function. In this section, we will cover two fundamental higher-order functions commonly used when working with arrays: **map** and **reduce**. Once you become familiar with these functions, you will find yourself using **for** and **while** loops much less often, and your code will become much neater.

For this section of the lab, we will not be using **let** and **const** to declare variables. This is only for convenience when experimenting in the browser console to avoid redeclaration errors. However, you should always scope your variables appropriately in .js script files!

5.1. MAP

Imagine that you have an array of numbers and wish to create a new array of numbers with 1 added to each element. Using a **for** loop, this would look something like:

```
oldNumbers = [1, 2, 3];
newNumbers = [];
for(let i = 0; i < oldNumbers.length; ++i) {
    newNumbers[i] = oldNumbers[i] + 1;
}
console.log(newNumbers);
```

However, you might notice that there are only two key pieces of information you need to specify to solve your particular problem: the input array and a transformation to apply to each element. The rest is essentially just boilerplate code to apply the transformation to each element of the input array.

The **map** function is a common concept available in many programming languages, which simplifies code of this form. It takes an input array and a transformation function, and returns a transformed array. So, using the **map** function, the above code simplifies to the following:

```
oldNumbers = [1, 2, 3];
newNumbers = oldNumbers.map(x => x + 1);
console.log(newNumbers);
```

5.1.1. EXERCISE 4

Rewrite the following code using **map**:

```
names = ['Alice', 'Bob', 'Cthulhu'];
greetings = [];
for(i = 0; i < names.length; ++i) {
    greetings[i] = 'Hello ' + names[i] + '!';
}
console.log(greetings);
```

5.2. REDUCE

map is useful for making one-to-one transformations of elements, but what if you want to take an array and produce some kind of aggregate value? For example, using a **for** loop, you might calculate the sum of values in an array like so:

```
myNumbers = [4, 7, 2];
sum = 0;
for(i = 0; i < myNumbers.length; ++i) {
    sum += myNumbers[i];
}
console.log(sum);
```

The **reduce** function simplifies code of this form. It accepts an input array, an accumulation function, and an initial accumulation value as arguments. In our case, the input array is **myNumbers**, the accumulation function is **(sum, x) => sum + x**, and the initial accumulation value is 0. Equivalent code using **reduce** looks like this:

```
myNumbers = [4, 7, 2];
sum = myNumbers.reduce((sum, x) => sum + x, 0);
console.log(sum);
```

An important thing to note is that **reduce** operates in left-to-right order through the array. This doesn't matter for our example above since addition is commutative. However, you may encounter instances where order matters, so it's worth keeping in mind.

5.2.1. EXERCISE 5

Use **reduce** to write code which calculates the maximum value in the array **myNumbers**. Hint: You can use **Math.max** to calculate the maximum of two numbers.

5.2.2. EXERCISE 6

Complete the following function for displaying the contents of an array on the web page, and enter it in the browser's JavaScript console.

```
displayArray = (inputArray) => {
    // Create an array where each element contains <li>item</li>
    let itemArray = inputArray.map(/* ...TODO... */ );
    // Concatenate all the elements of list into one string
    let oneString = itemArray.reduce(/* ...TODO... */ );
    document.getElementById('cool-section').innerHTML =
        '<ul>' + oneString + '</ul>';
}
```

Test the function by calling it on the greetings array you created in the exercise from section 5.1. Look at the yellow part of the page to see what happens.

6. LODASH

Lodash is a JavaScript library that contains many useful functions, making programming a lot more pleasant. Lodash has been included in this web page (open **index.html** in a text editor to find the line of HTML that does this). The official documentation at https://github.com/lodash/lodash/tree/4.13.1/doc provides a full list of the functions provided by Lodash, along with explanations. Here, we will cover some of the more useful functions.

6.1. _.REJECT

_.reject takes an array and a predicate function as input. A predicate function is simply a function that returns true or false. The predicate function is run on all elements of the input array, and a new array is produced that contains only the elements for which the function returned false.

For example, here is how we would use **_.reject** to remove all elements in an array that are less than 5:

```
oldNumbers = [4, 7, 2];
newNumbers = _.reject(oldNumbers, x => x < 5);
console.log(newNumbers);
```

6.2. _.FINDINDEX

_.findIndex takes an array and a predicate function as input and returns the index of the first element for which the predicate is true.

```
heroes = ['The Hulk', 'Wonder Woman', 'Batman'];
_.findIndex(heroes, hero => _.endsWith(hero, 'man'));
```

If the end of the array is reached before the predicate returns true, **_.findIndex** will return -1.

```
_.findIndex(heroes, hero => _.endsWith(hero, 'merica'));
```

6.2.1. EXERCISE 7

Use **_.findIndex** to return the index of the element which is exactly equal to 'Batman' in the heroes array.

6.3. _.ASSIGN

_.assign is an incredibly useful function for updating objects and arrays. It takes all of its arguments after the first and merges them into the first argument. The only argument modified is the first, so providing an empty object or array means that you will not modify any of your existing data.

For example, let's say that you want to create a new copy of an object with one of the fields updated. No problem!

```
old_record = { name: 'Bob', age: 20 };
new_record = _.assign({}, old_record, { age: 21 });
```

How about adding a new field? Assign has you covered!

```
new_record = _.assign({}, old_record, { mood: 'Curious' });
```

You can also do both at once.

```
new_record = _.assign({}, old_record, { age: 21, mood: 'Curious' });
```

You can work with arrays in a similar way:

```
items = ['Orange', 'Tomato', 'Banana'];
new_items = _.assign([], items, { [1]: 'Apple' });
```

7. PROMISES

All of the code that we have written thus far has been sequential. That is, the instructions are executed one at a time and in order. However, sometimes you will need to perform an action that takes a long time to complete, such as downloading an image or performing a database query. Since these operations are slow, you will usually want to allow other code to run while they are being completed.

Promises provide a mechanism for handling asynchronous operations in JavaScript. Instead of calling a function and getting the result returned when it is ready, the function will immediately return a promise to deliver the result at a later time. The promise starts off in a "pending" state and will eventually either be "rejected" or "fulfilled".

Try running the following image-downloading code to see what I mean:

```
// This will just output a Promise with state "pending"
console.log(fetch('https://httpbin.org/image/jpeg'));
```

We can then continue running whatever other code we want while the image is downloading.

"But how do I get the downloaded image?" you may ask. Well, we need to provide a success callback function to the promise using the **.then** method. This effectively says, "Hey, when the result is ready (the promise is fulfilled), I want to run the code in this callback function." The callback function can also return another promise, enabling the chaining of multiple operations.

Here is an example of using promises to download an image and add it to the bottom of the current web page:

```
// This will wait for the returned Promise to resolve, and do
// something with the response.
fetch('https://httpbin.org/image/jpeg') // Start a download
    .then(res => res.blob()) // Get the content blob
    .then(blob => { // Use the content blob
        const img = '<img src="' + URL.createObjectURL(blob) + '">';
        document.body.insertAdjacentHTML('afterend', img);
    })
    .catch(err => console.error(err));
```

You may notice that we also used the **.catch** Promise method to provide a failure callback. Sometimes an operation may fail, for example, if the Internet connection drops mid-download. In such cases, we say that the promise was rejected. The success callbacks will be skipped, and the failure callback will be called with an error object indicating what went wrong.

8. [BONUS] PUTTING IT ALL TOGETHER

Wow, that's a lot of information! Hopefully you can draw a lot of similarities to programming knowledge that you have already established, as that makes learning a new language much easier.

8.1. EXERCISE 8

[Bonus] Now it's time to apply your JavaScript skills to a simple programming problem. The task is to write a function called **createCircle** which produces circle objects with two methods: **area** and **perimeter**.

Let's take a look at what we want to be able to do once you've written your code.

```
// INPUT
const circle = createCircle(4);
// OUTPUT
undefined

// INPUT
circle
// OUTPUT
{ radius: 4, area: [Function], perimeter: [Function] }

// INPUT
circle.perimeter()
// OUTPUT
25.132741228718345

// INPUT
circle.area()
// OUTPUT
50.26548245743669
```

So **createCircle** accepts one parameter, a radius, and returns a circle object with that radius and methods for calculating the perimeter and area of the circle. The value of pie is available as **Math.PI**. Hint: recall that arrow functions are not good for creating methods.

8.2. EXERCISE 9

[Bonus] Build upon the previous code to write a function called **createCircles** which takes an array of radii as input and returns an array of circles as output. Hint: the **map** function is your friend.