Exercises: More Advanced JavaScript

We will once again work in the REPL and you do not need to install anything.

As before, you can go to this link for direct access to a Node.JS repl. If there are issues loading this one – sometimes you may hit capacity limitations - try the Babel REPL.

Arrays

Similar to Lists in Python and arrays in C, C# and other languages, JavaScript arrays are used to store multiple values in a single variable. They are a fundamental data structure and their syntax and operation is very similar to C and to a lesser extent like C# and Java.

Using an array literal is the easiest way to create a JavaScript Array:

```
let array_name = [item1, item2, ...];
```

Spaces and line breaks are not important. A declaration can span multiple lines:

```
let animals = [
 "Lion",
 "Tiger",
 "Giraffe",
 "Elephant",
 "Monkey",
 "Lemur",
 "Rhinoceros"
];
```

The most important thing to remember about arrays in JavaScript is that they are zero-indexed: ***array indexes start with 0. [0] is the first element. [1] is the second element.***

This statement accesses the value of the first element in animals:

```
let firstAnimal = animals[0]; // Will be "Lion"
```

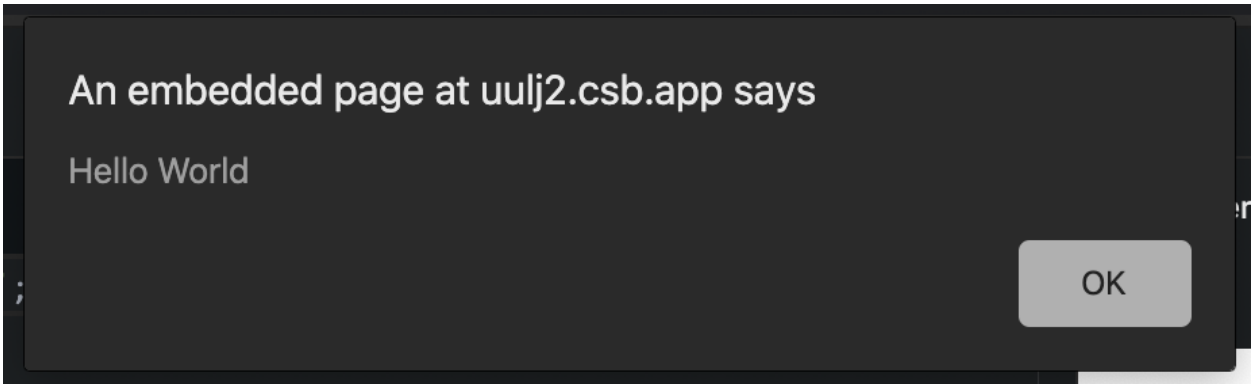
Arrays can contain basically any valid JavaScript expression such as strings, numbers, arrays, objects, even functions. Unlike in strongly typed languages such as C# and Java, this mixing of element types is legal in JavaScript. The very weird array that you see on the next page actually includes a function declaration as one of the elements.

```
const myWeirdArray = [
 "Text goes here",
```

```
 1,
 [0, 1, 2, 3],
 function myFunction() {
   alert('Hello World')
 }
];
```

As usual, all of these items work the same as they would in a normal variable if you use the expression to access that array item instead of the variable name. E.g. to call the above function `myFunction` refer to its position in the array followed by parentheses '()'.

```
myWeirdArray[3]()
```



Arrays in JavaScript also support many useful, standard properties and methods such as `length`, `includes()`, `push()`, `slice()` and so on. Many more can be found here. We will generally use these as we need to, explaining them in context, but most of them will be familiar from other languages.

console.log & console.table

We have implicitly worked with the `console` from the start, but here we talk about it a bit more. We use it mainly for simple testing purposes. Logging variables using `console.log()` shows the value of the variable at that point in your code, making it easy to ensure it is doing what it should:

```
console.log(animals);
```

```
▼(7) ["Lion", "Tiger", "Giraffe", "Elephant", "Monkey", "Lemur",
"Rhinoceros"] i
   0: "Lion"
   1: "Tiger"
   2: "Giraffe"
   3: "Elephant"
   4: "Monkey"
   5: "Lemur"
   6: "Rhinoceros"
   length: 7
 ▶ __proto__: Array(0)
```

Console `table` outputs the variable in a table format in the console which can be a nicer way of viewing arrays and objects:

```
console.table(animals);
```

VM139:1

(index)	Value
0	"Lion"
1	"Tiger"
2	"Giraffe"
3	"Elephant"
4	"Monkey"
5	"Lemur"
6	"Rhinoceros"

Arrays are a special type of object. The `typeof` operator in JavaScript returns "object" for arrays.

```
console.log(typeof animals) // Prints "object"
```

But, JavaScript arrays are best described distinctly as they have different uses and syntax. In particular, we seldom use indexing styles when accessing objects.

Objects

JavaScript objects are generally defined in what is called 'JavaScript Object Notation' (JSON). JSON has become a standard format for transferring data which we will use a lot in later weeks. The basic JSON syntax is to have curly brackets (braces) `'{}'` surrounding the object's values. The values are written as key: value pairs (key and value separated by a colon).

```
const object_name = {
 key1: "Value1",
 key2: "Value2"
};
```

Arrays use index numbers to access their "elements" as shown in the examples above. Objects, however, use the key or property name to access their "properties" `(object_name.key1)`. In this example, `animal.name` returns Lion:

```
const animal = {
 name: "Lion",
 number: 3
};

console.log(animal.name) // Prints "Lion"
```

This is the preferred way of accessing object properties. You may encounter external data that has badly chosen keys containing spaces, periods or characters that this format doesn't support. In these cases the more array-like syntax of `animal["name"]` is also valid.

Because each property has its own name, objects allow us to provide context around the elements of data. Arrays are better in situations where there is a list of similar elements. If you have multiple numbers, books, or students where the difference between each item doesn't need to be explained - each element of the list is very much of the same type - then an array should be used. Objects are useful for more varied data that explains more information about an entity or piece of data.

In JSON data, arrays and objects are frequently used inside each other to have arrays of objects and object properties that are arrays. This can be confusing at first glance, but pretend that you are a compiler and start matching up braces and square brackets and it will quickly become clear:

```
const animalData = [
 {
   name: "Lion",
   number: 3,
   eats: ["zebra", "antelope", "buffalo", "hippopotamus"]
 },


 {
   name: "Tiger",
   number: 5,
   eats: ["moose", "deer", "buffalo"]
 },
```

```
 //...
 {
   name: "Lemur",
   number: 15,
   eats: ["fruit", "leaves", "insects"]
 },
 {
   name: "Rhinoceros",
   number: 2,
   eats: ["grass", "shoots", "leaves", "berries"]
 }
];


console.log(animalData);
console.table(animalData);
```

Unsurprisingly, we are very keen to save JSON and to load it from external sources. When we want to pass it around in a convenient format, it is usual to convert a JSON object to a string, and to parse these strings to recover the JSON object. Fortunately JavaScript helps us out:

`JSON.stringify()` - The `JSON.stringify()` method converts a JavaScript object or value to a JSON string.

```
JSON.stringify(animalData)
```

```
'[{"name":"Lion","number":3,"eats":["zebra","antelope","buffalo","hippopotamus"]},{
"name":"Tiger","number":5,"eats":["moose","deer","buffalo"]},{"name":"Lemur","numbe
r":15,"eats":["fruit","leaves","insects"]},{"name":"Rhinoceros","number":2,"eats":[
"grass","shoots","leaves","berries"]}]'
```

`JSON.parse()` – The `JSON.parse()` method parses a JSON string into a JavaScript object. This can be necessary when receiving data from APIs and other sources.

In our final JavaScript topic we will look at a series of methods that sit on top of arrays, allowing us to process data in a functional style. To make this code more compact, we will often use the JavaScript arrow notation for the declaration of anonymous functions. The terms and the syntax seem challenging, but they are not that bad once you understand the purpose. Sometimes we want a function to do some work for us in a specific context - here we will use it to filter, select or transform the elements of an array. Such functions are normally trivial, and we don't want to use them again. In modern languages we can integrate the functions more closely with the code and not even bother to give them a name. These are what we call anonymous functions, and many languages have a special syntax for them. This is considered below.

Arrow and Anonymous functions

In the example below, we will show you different ways to write a function to double a number - so that if the value of `num` is passed as 4, the function will return a value of 8.

All of the following examples are valid ways to write this doubling function. In each case, on the left hand side we will provide a template format showing the style of declaration. On the right hand side, we will see the corresponding example.

So here is the basic function we are familiar with: it has a function keyword, the name of the function, and parentheses surrounding any parameters and curly braces around the function body.

```
function name (parameters) {
  statements
}
```

```
function double (num) {
  return num * 2;
}
```

As you saw earlier, we can also declare a function without a name, assigning it to an identifier as shown below. We won't consider scope here, but the rules still apply. Note the use of `const` rather than `let` - once we assign a function to a variable we seldom wish to change it to something else.

```
const name = function (parameters) {
  statements
}
```

```
const double = function (num) {
  return num * 2;
}
```

This formal way of defining an anonymous function is often used in parameter lists and elsewhere, but eventually someone found it tiresome and proposed a shorter syntax.

We now simplify our declaration to get an *arrow function*. This is done by removing the `function` keyword and instead putting an arrow like syntax `=>` in between the parameters and the function body. This works exactly the same as in the example above, but it is widely seen as a more convenient way to declare anonymous functions.

```
const name = (parameters) => {
  statements
}
```

```
const double = (num) => {
  return num * 2;
}
```

If it is a basic function that only needs one line for a return statement, we don't even need the curly braces or return keyword. We can write the same thing like this:

```
const name = (parameters) => expression
```

```
const double = (num) => num * 2;
```

Note: on the left 'statement' has been changed to 'expression' because it now has to be a single valid JavaScript expression that will be returned.

This syntax makes it really easy to write a simple compact function if all it needs to do is return an expression. This is particularly common in higher order functions like map, filter and reduce. It may be a very small amount of code and not look much like a function but it is still its own function.

Map, Filter, Reduce

Map, Filter and Reduce are useful array methods that are known as higher order functions. Higher-order functions are functions that take other functions as arguments or return functions as their results. The resulting code is said to adopt a functional style.

Map

The `map()` function creates a new array populated with the results of calling a provided function on every element in the calling array. The map function is used extensively in React to generate JSX from an array. Here we show a simple example where the anonymous arrow function from before is used to double every element in the specified array:

```
const array1 = [1, 4, 9, 16];

// pass a function to map
const map1 = array1.map(x => x * 2);

console.log(map1);
// expected output: Array [2, 8, 18, 32]
```

Filter

The `filter()` method creates a new array with all elements that pass the test implemented by the provided function.

Here is a simple example where each word in the array is tested to see if it has a length of greater than six characters.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

Reduce

The `reduce()` method executes a reducer function (that you provide) on each element of the array, resulting in a single output value. This may appear more confusing than the others. The second parameter of the reduce method is an initial value for the 'accumulator'.

In the example below, the initial value given is 5, and the accumulator becomes the additive total of the numbers in the array.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Later on in the semester, leading into assignment 1, we will provide you with some guidance on connecting your JavaScript knowledge with your need to write code that works on client side web pages. Until then we suggest that you work mainly with our prac examples, which will gradually introduce you to node and ask you to do more and more on the server side.