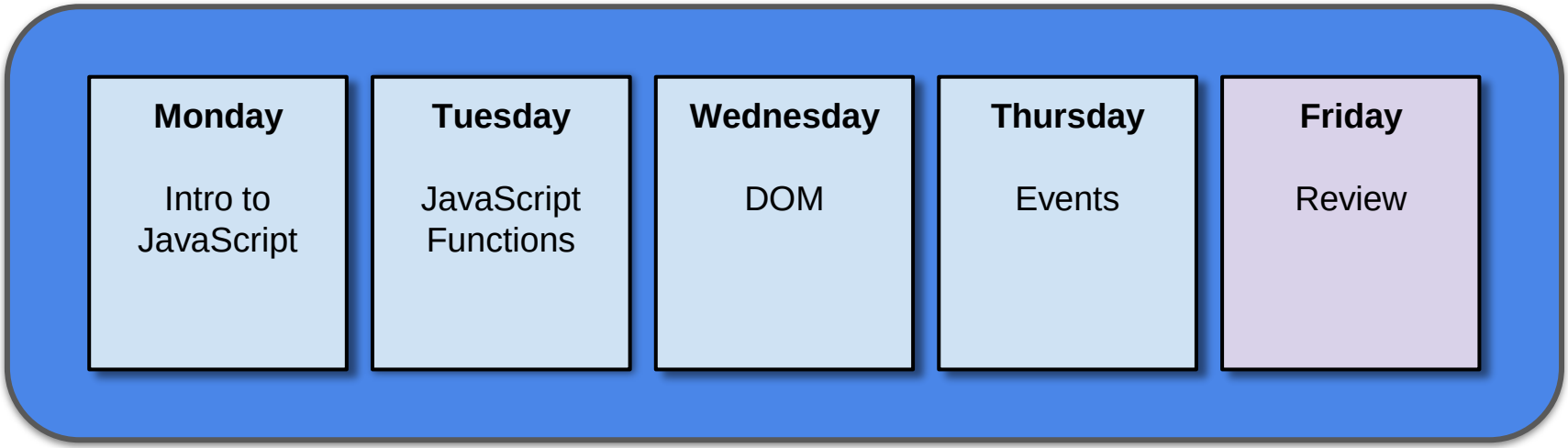
Introduction to JavaScript

Module 3: 06

Week 10 Overview



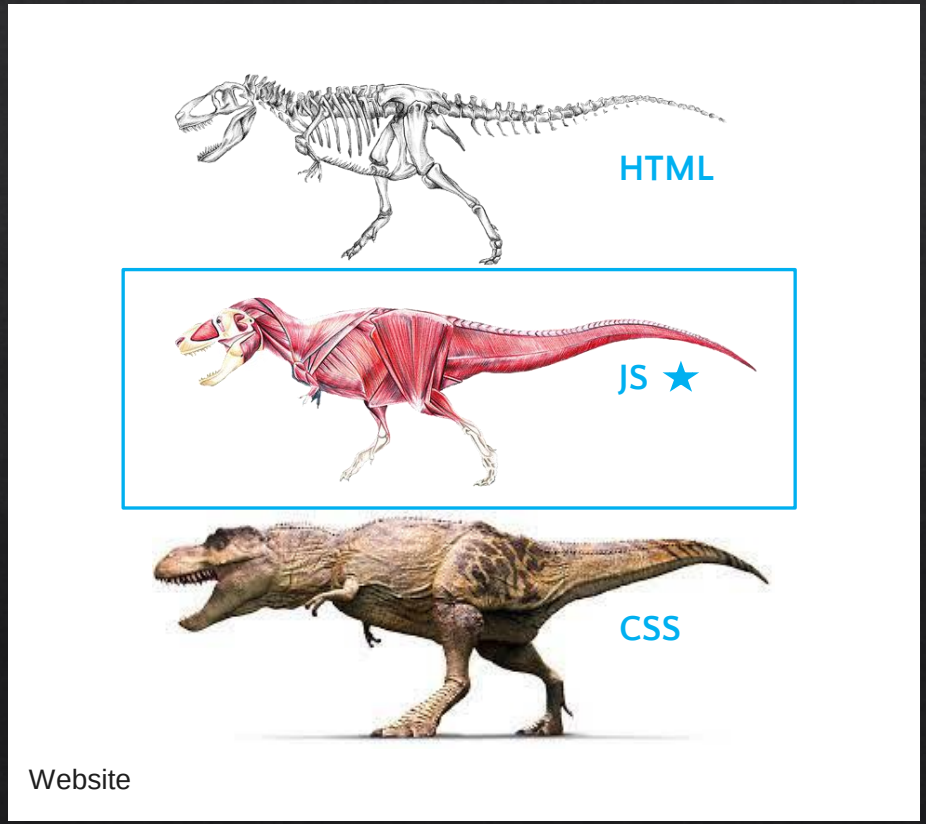
Today's Objectives

1. Client Side Scripting
2. Intro to JavaScript
3. Including JavaScript
4. JavaScript Overview
    a. Variables in JavaScript
    b. Load Order and Hoisting
    c. Data Types
    d. Dynamic Typing
    e. Named Functions
    f. Equality Operators
    g. Truthy
    h. Scope
    i. Arrays
    j. Object Literals
    k. Numbers, Math, Date, and String Libraries and Methods
5. Running Unit Tests and Debugging

Client Side Scripting

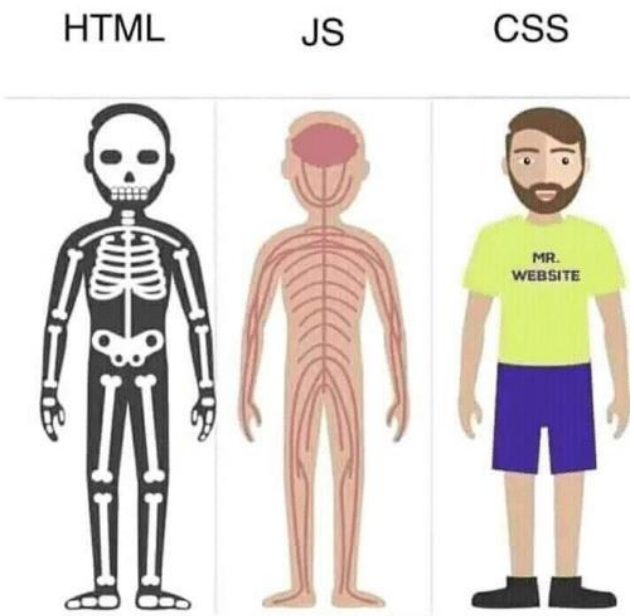
- Executes code in the user's browser, allowing our code to interact with the rendered HTML and CSS
- Creates less stress on the server, and a better experience for the user
- Uses
  - Responding to an Event (mouse click, key press, resize, etc.)
  - Interact with APIs to dynamically update a page
  - Manipulate the loaded page without needing to refresh

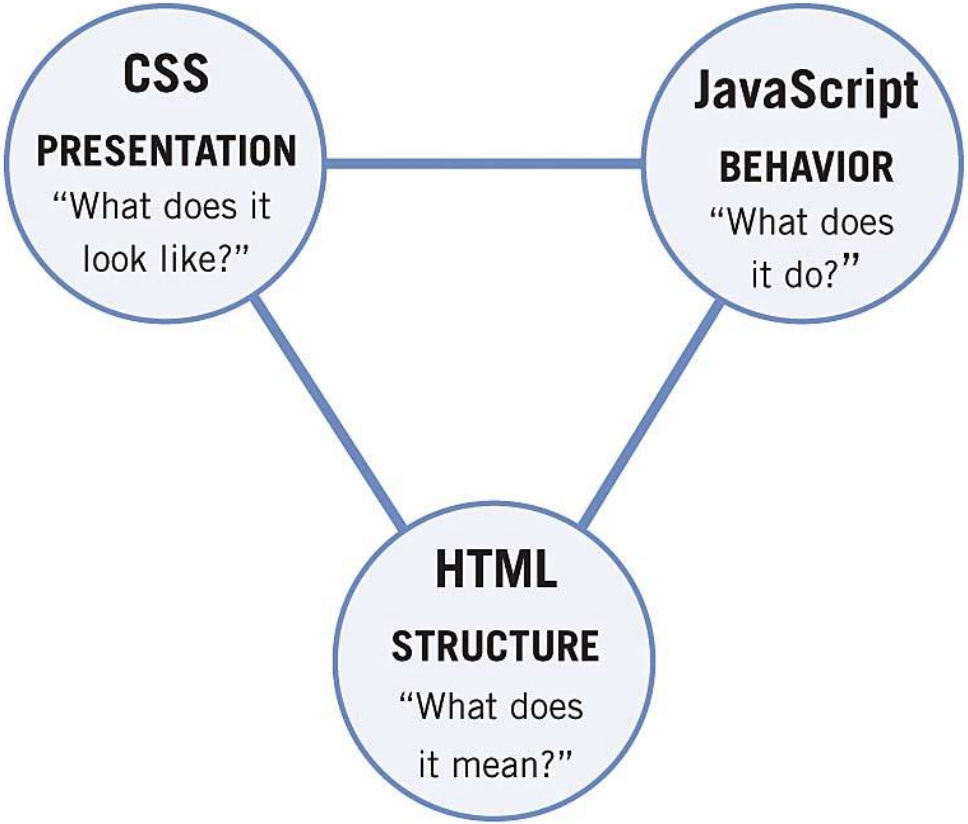
HTML / CSS / JS



Client Side Scripting - Separation of Concerns

1. **HTML** - provides content and structure
2. **CSS** - provides presentation
3. **JavaScript** - provides behavior





JavaScript: Where does It Go?

JavaScript can be incorporated directly into a HTML page:

```
<html>
<head>
     <script>
          window.alert('Hello World.');
     </script>
</head>
<body>Helpful Content.</body>
</html>
```

A block of JavaScript code is enclosed in a set of <script> tags.

JavaScript: Where does It Go? (Preferred Method)

It is recommended that JavaScript logic be placed in a separate file and “included” in the HTML file.

```
<html>
<head>
<script src="thisScript.js"></script>
</head>
<body>Helpful Content.</body>
</html>
```

```
window.alert('Hello World.')
```

This is preferred over the first method we discussed.

Loosely Typed

- In terms of data types, JavaScript is loosely typed, meaning we do not explicitly tell JavaScript what data type a variable will hold.
- These are the data types a variable can take on: **String**, **Number**, **Boolean**, **Arrays**, and **Objects**.

```
typeof 1 //> "number"
typeof "1" //> "string"


typeof [1,2,3] //> "object"
typeof {name: "john", country: "usa"} //> "object"


typeof true //> "boolean"
typeof (1 === 1) //> "boolean"


typeof undefined //> "undefined"
typeof null //> "object"


const f = () => 2
typeof f //> "function"
```

Declaring Variables

- Declaring variables in JavaScript takes on the following form:

  let <<variable name>> = <<initial value>>;

```
let myStrVariable = 'hammer';
let myNumVariable = 3;
let myOtherNumVariable = 3.14
let myBoolean = true;
```

- In older texts you will see variables declared using var, i.e. var myBoolean = true. This should be avoided at all costs, **always use let**.
- Values that do not change are declared using **const**.

typeof

- We can use typeof to ascertain the data type of a variable.

```
let myStrVariable = 'hammer';
console.log(typeof myStrVariable); // string
let myNumVariable = 3;
console.log(typeof myNumVariable); // number
let myOtherNumVariable = 3.14
console.log(typeof myNumVariable); // number
let myBoolean = true;
console.log(typeof myBoolean); // boolean
```

Conditional Statements and Comparisons

These should also look familiar:

```
let x = -3;
let positive = (x > 0);
console.log(positive);
// Prints false

if (x < 0) {
   console.log(x + ' is a negative number.');
}
// Prints -3 is a negative number
```

Conditional Statements and Comparisons

We can also apply AND / OR / XOR statements:

```
let x = -3;
let y = -4
let positive = (x > 0);

if (x < 0 && y < 0) {
   console.log('Both numbers are negative.');
}
else if ( x < 0 ^ y < 0) {
   console.log('Only one is negative.');
}
else {
   console.log('Both are positive');
}
```

Declaring An Array

Here are a few examples of array declarations:

```
//Declaring an array with three strings:
let myArray = ['Fiat Chrysler', 'Ford', 'GM'];

//An empty array:
let myEmptyArray = [];
```

Iterating Through an Array

Our loopy friends are back:

```
let myArray = ['Fiat Chrysler', 'Ford', 'GM'];
for (let i=0; i < myArray.length; i++) {

   console.log(myArray[i]);
}
// Prints out Fiat Chrysler Ford GM
```

- Note that the for-loop is structurally similar to its Java counterpart.
- We access individual elements of an array in a similar way: myArray[0] for the first, myArray[1] for the second, etc.
- We can access the length of an array with the .length property.

Comparison operators

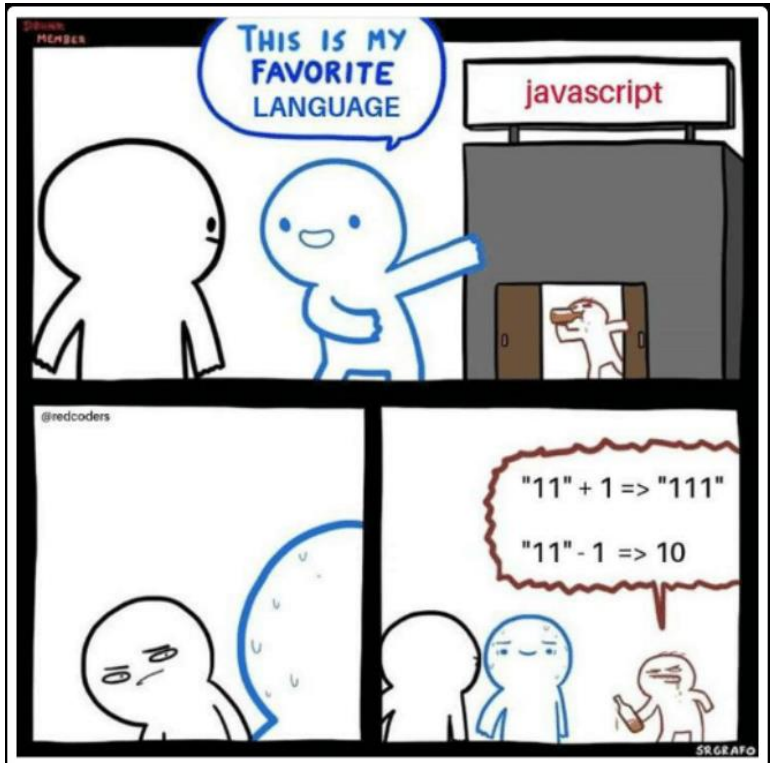
Identity vs. Equality:

```
let x = 10;
if (x === '10'){

   console.log("equal");
}
// x is a number and '10' is a string so this is
false
```

- Identity (===) operator behaves identically to equality (==) operator except no type conversion is done
- == operator will compare for equality after doing any necessary type conversions
- === will not do the conversion and will only consider the two equal if they are of the same type and same values

Truthy and Falsy



If you are coming from a strictly typed language like Java, there are some unusual things to consider with regards to data type, one of these is the idea of “truthy” and “falsy.”

Truthy and Falsy

- These rules can sometimes strike one as bizarre, but we can derive an intuitive understanding of what's going on. Here is a good site with a more in-depth explanation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness
- For now, consider the following code:

```
let i ='1';
let j = 1;
console.log(i == j); // true
console.log(i === j); // false
```

- The triple equals is to evaluate "strict equality" - meaning that not only do the values have to be the same, but the types must equal as well.

Truthy and Falsy

The following values are **always falsy**:

- `false`
- `0` (zero)
- `''` or `""` (empty string)
- `null`
- `undefined`
- `NaN`

Everything else is **truthy**. That includes:

- `'0'` (a string containing a single zero)
- `'false'` (a string containing the text “false”)
- `[]` (an empty array)
- `{}` (an empty object)
- `function(){}` (an “empty” function)

```
function testTruthyFalsy (val)
{
    return val ? console.log('truthy') : console.log('falsy');
}

testTruthy(true);                   // truthy
testTruthy(false);                  // falsy
testTruthy(new Boolean(false));     // truthy (object is always true)

testTruthy('');                     // falsy
testTruthy('Packt');                // truthy
testTruthy(new String(''));         // true (object is always true)

testTruthy(1);                      // truthy
testTruthy(-1);                     // truthy
testTruthy(NaN);                    // falsy
testTruthy(new Number(NaN));        // truthy (object is always true)

testTruthy({});                     // truthy (object is always true)

var obj = { name: 'John' };
testTruthy(obj);                    // truthy
testTruthy(obj.name);               // truthy
```

Objects

- JavaScript is not generally considered an object oriented language, it is instead a functional language (one that is based on functions).
  - Over time though, some OO features have been added to the language.

- JavaScript objects follow JSON notation, with the object itself surrounded by curly braces, and the object properties listed in comma delimited key-value pairs:

  { **prop1**: **<<prop1Value>>**, **prop2**: **<<prop2Value>>**}

Objects Example

Let’s look at a concrete example:

```
let crewMember = {
firstName: 'James',
lastName: 'Kirk',
rank: 'Captain'
};

console.log(crewMember.firstName);
console.log(crewMember.lastName);
console.log(crewMember.rank);
crewMember.rank = 'Admiral';
console.log(crewMember.rank);
```

Loosely Typed

- In terms of data types, JavaScript is loosely typed, meaning we do not explicitly tell JavaScript what data type a variable will hold.
- These are the data types a variable can take on: **String**, **Number**, **Boolean**, **Arrays**, and **Objects**.

```
typeof 1 //> "number"
typeof "1" //> "string"


typeof [1,2,3] //> "object"
typeof {name: "john", country: "usa"} //> "object"


typeof true //> "boolean"
typeof (1 === 1) //> "boolean"


typeof undefined //> "undefined"
typeof null //> "object"


const f = () => 2
typeof f //> "function"
```

JavaScript- Another Look - History

- Developed at Netscape Communications by Brendan Eich in 1995 in 10 days as a way to make web pages more dynamic
- Syntax and functionality inspired by C, Java, Scheme, and Self
- At the time web development was not considered “serious” development
- Originally named Mocha then LiveScript, changed to JavaScript as part of a marketing deal with Sun (who owned Java)
- Hated and misunderstood until the mid-2000s when “rediscovered” as an easy way to use AJAX and JSON with Web APIs
- Standardized and versioned as ECMAScript, currently ECMAScript 6
- Currently the de facto language of web development

Java vs JavaScript

Java	JavaScript
Syntax based on C	Syntax based on C
Object Oriented	Object Based (prototypes)
Compiled	Interpreted
Run by the JVM/JRE	Runs in a browser (engine)
Strict syntax	Loose Syntax
Statically Typed	Dynamically typed
Everything is a data type	5 data types

Including JavaScript

Embedded in a Script Tag.

```
<html>
  <head>
    <script type="text/javascript">
      console.log('Hello JavaScript!');
    </script>

    <!-- Other Nodes omitted -->
  </head>
  <body>
    <!-- Content omitted -->
  </body>
</html>
```

The <script> tag can be included anywhere in the head or body of the html document.

<script> is not self-closing, so must always have an ending tag: <script></script>

Script Blocks

.JS Files

JS File Reference

```
<html>
  <head>
    <script type="text/javascript" src="myJsFile.js"></script>
    <script type="text/javascript" src="https://someurl.com/code.js"></script>

    <!-- Other Nodes omitted -->
  </head>
  <body>
    <!-- Content omitted -->
  </body>
</html>
```

Variables in JavaScript

var - Old way of defining variables. ***DO NOT USE!!!***

```
var name = 'Rachelle';
```

let - new way of declaring variables. The value can be reassigned. *For values that can change.*

```
let name = 'Rachelle';
name = 'John';    ← No Error
```

```
let name;
name = 'Matt';    ← No Error
```

const - new way of declaring variables. The value must be assigned at declaration and cannot be reassigned. *For values that will not change - this should be your default.*

```
const name = 'Rachelle';
name = 'John';    ← Error
```

JavaScript Semantics

1. Uses single quotes (') preferred, but can use double quotes
2. Can also use tick (`) for template literals
    a. Template Literal string that has a variable as a placeholder
        i. name = 'Joe' → 'Hello, ' + name + ' welcome!' → 'Hello, Joe welcome!'
        ii. name = 'Joe' → `Hello, ${name} welcome!` → 'Hello, Joe welcome!'
3. Semicolons at the end of a statement are optional, but preferred for readability.
4. Blocks of code are defined with { }
5. Functions are called with ( )
6. variables and functions use camelCase

Data Types

JavaScript has 5 data types

1. string

```
let name = 'John';
```

2. number

```
let favoriteNumber = 42;
```

3. boolean

```
let isDynamicallyTyped = true;
```

4. object

```
let person = { name: 'Rachelle', numberGiantClocks = 1 };
```

5. function

```
let logFunction = console.log('test');
```

Dynamic vs Static Typing

A **statically typed** language enforces data type constraints at compile time, so variable must be declared with a data type, and then can only hold data of that type. *Java is a statically typed language.*

```
String name = "Rachelle";

name = 10;   ← error, because name can only hold a string
```

A **dynamically typed** language infers the data type of the variable from the value it hold at run-time. Variables are not declared with a type and can hold any value. *JavaScript is a dynamically typed language.*

```
let name = "Rachelle";   ← name is now a string

name = 10;   ← now name is a number
```

Defining Named Functions

- defined with the word function
- lack a return a return type
- have no access modifier
- parameters do not have defined data types

```
1 function addNumbers(x, y) {
2   const sum = x + y;
3   return sum;
4 }
```

Calling Functions

- parameters defined by order
- not enough parameters = undefined for the extras parameters
- Too many parameters = extra values are ignored.

```
1 function addNumbers(x, y) {
2   const sum = x + y;
3   return sum;
4 }
5
6 const result = addNumbers(60, 6);
```

```
addNumbers(10)
```

x = 10

y = undefined

```
addNumbers(10, 20, 30, 40, 50)
```

x = 10

y = 20

30, 40, 50 are ignored

Equality Operators (== vs ===)

== (Loose equality)

- compares the value without taking the data type into consideration

1 == ‘1’ → true
1 == 1 → true
1 != 1 → false (not equal)

=== (Strict Equality)

- compares the value and the data type

1 === ‘1’ → false
1 === 1 → true
1 !== 1 → false (not equal)

JavaScript is a Truthy Language

- Everything in the language evaluates to either true or false
- falsy (evaluate to false): false, 0, ‘’, null, undefined
- truthy (evaluate to true): everything else

Truthy values

```
'false' (quoted false)
'0'     (quoted zero)
()  (empty functions)
[]  (empty arrays)
{}  (empty objects)
All other values
```

Falsey values

```
false
0    (zero)
''   (empty string)
null
undefined
NaN
```

Truthy

```
1 const favoriteNumber = 42;
2
3 if (favoriteNumber) {
4   console.log('This will be logged');
5 } else {
6   console.log('This will not be reached since 42 is truthy');
7 }
```

```
 1 const result = calculate(); // Might be null or undefined
 2
 3 if (result) {
 4   // Wasn't null / undefined
 5 } else {
 6   // Was null, undefined, or otherwise falsey
 7 }
 8
 9 // Alternative way of writing this could be:
10 if (result !== null && result !== undefined) {
11   // ...
12 }
```

Scope

- Global Scope
  - variables defined outside of a function are available everywhere, even in other js files that are included in the html
- Block Scope
  - works like block scope in Java
- Function Scope
  - variable can be used anywhere in a function that it is declared in
- let and const - obey block and function scope
  - Only allow 1 declaration of a variable per scope (like Java)
- var - ignores block scope, and obeys function scope
  - A variable can be redeclared in the same scope

Function Scope

```
function add(x, y) {
        // function scope
}
```

Obeyed by let, const, var

Block Scope

```
if (x === 10) {
        // block scope
}
```

Obeyed by let, const
Ignored by var

Arrays

- defined by []
  - let x = []; ← declared an empty array
  - let x = [1, 2, 3, 4]
- Not confined to a single data type
  - let x = [1, 'a', 2]
  - Not a fixed size
  - x[3] = 'b'; → [1, ‘a’, 2, ‘b’]
- Methods *(not a complete list)*
  - length()
  - concat() ← joins to arrays into 1,
  - push() → adds an element to the end of an array,
  - pop() → removes the last element from an array
  - unshift() → adds an element to the start of the array

Object Literals

- defined with keys and literal values as being declared
- Objects assigned to a variable
- Defined as a *comma delimited* block of **key : value** pairs
- let x = {} ← declares an empty object

```
const person = {
  firstName: "Bill",
  lastName: "Lumbergh",
  age: 42,
  employees: [
    "Peter Gibbons",
    "Milton Waddams",
    "Samir Nagheenanajar",
    "Michael Bolton"
  ]
};
```

Math and String Functions

- isNaN() - returns true or false if a variable is not a number
- Math
  - Built in Math object that has properties and methods for mathematical operations
    - Math.PI, Math.abs(), Math.floor(), Math.random()
  - Documentation on MDN
- String
  - The String data type has properties and methods to manipulate the value of the String
    - .length
    - .endsWith(), .startsWith()
    - .indexOf()
    - .split()
    - toLowerCase(), .toUpperCase()
    - .substr(startIndex, numberOfCharactersToReturn)
    - .substring(startIndex, endingIndex)
    - .trim()
  - Documentation on MDN

Math and String Functions

- String
  - Example:
    - .startsWith()

```
let x = 'aaabbbc’;
if (x.startsWith('aaa')){ console.log('yes')}
```