JavaScript Fundamentals

- Just as HTML provides structure and content to a web page and CSS provides style and layout, JavaScript (or just JS) provides ***interactivity*** and ***logic*** to a web page.
    - JS is as integral to building a modern web page as are HTML and CSS.

- You can do a lot with JS:
    - Manipulate HTML and CSS
    - React to user actions and other events
    - Geolocate
    - Create 2D and 3D graphics
    - Use multimedia
    - Use 3rd party APIs
        - Twitter
        - Google Maps
        - Flickr
        - Many, many others

- JS is very different than the programming languages you've learned up until now (C++ and C), but it also has many similarities.

- Here, we'll give you a crash course on JS, assuming that you already know how to program with C++ and/or C and paying special attention to the places JS is different than those languages.

JS Fundamentals

- JS is typically thought of as an ***interpreted*** language.
    - An interpreted language is not compiled. Code is run as-is, line by line, from top to bottom, and the result of each line is immediately returned.
    - Most modern browsers actually *do* compile JS code using just-in-time (JIT) compilation.
        - JIT compilation is compilation at runtime, during the execution of a program.
    - Each browser has its own JS interpreter/engine:

        - V8: Chrome, Opera
        - SpiderMonkey: Firefox
        - JavaScriptCore/Nitro: Safari
        - Chakra: Internet Explorer, MS Edge

- JS is a ***loosely-typed*** language: variables are not assigned a specific type in the code, but values have some notion of type when instantiated.

- JS has ***first-class functions***: functions themselves are treated as data.
    - Functions can be passed as arguments to other functions.
    - Functions can be returned from other functions.
    - Functions can be assigned to variables or stored in data structures.

Adding JS to your webpage

- There are two main ways (and a third, unrecommended way) to add JS to a web page:
    - Internal JS
    - External JS

- Internal JS is written directly within the HTML, inside a `<script>` element:

```
<script>
     // JS goes here.
</script>
```

- External JS is written in a separate file and included (similar to CSS) using `<script src="...">`:

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

- In both cases, the `<script>` element is typically placed within the `<head>` element.

JS variables

- In JS, a variable is declared using the `var` keyword:

```
var v;
```

- Variables in JS are case-sensitive and are typically written in `lowerCamelCase`.

- There are a few essential types of data we can store:
    - Numbers:

      ```
      var i = 5;
      var f = 3.1415;
      ```

    - Strings:

      ```
      var str = 'abc';
      ```

    - Booleans:

      ```
      var t = true;
      var f = false;
      ```

    - Arrays:

      ```
      var arr = [ 1, 2, 3 ];
      ```

    - Objects:

      ```
      var obj = { name: 'Francis', species: 'cat' };
      ```

    - Functions:

      ```
      var fn = function() {};
      ```

- We can inspect a variable's type using `typeof()`:

```
typeof(v);
```

Printing to the console in JS

- In JS, we use the `console.log()` method to print things to the console in the browser's developer tools, e.g.:

  ```
  console.log("This will be printed to the console.");
  console.log("This is the value of str:", str);
  console.log("This is the value of obj:", obj);
  ```

Numbers in JS

- All numbers in JS, both integers and floating-point numbers, have the same type: `Number`.

- Arithmetic operators (`+`, `-`, `*`, `/`, `%`) are, for the most part, the same as in C++ or C.
  - One caveat is that division is always floating-point division:

    ```
    var n = 9 / 5;  // This will be 1.8.
    ```

Comparisons

- Comparisons are slightly different in JS.

- We typically use the strict equality (`===`) and strict inequality (`!==`) operators.

- Simple equality (`==`) and inequality (`!=`) disregard data type and can give unexpected results, e.g.:

  ```
  2 === '2'  // This will be false.
  ```

  Is different than

  ```
  2 == '2'  // This will be true.
  ```

Strings

- Strings can be written in `'single quotes'` or in `"double quotes"`.
  - The beginning quote must match the ending quote.

- Strings can be concatenated using the + operator:

  ```
  var str = "hello, my name is " + "Rob";
  ```

  - You can coerce different data types to strings using +:

    ```
    var str = "the number is " + 8;
    ```

- To determine the number of characters in a string, use `str.length`

- You can access the characters of a string using array notation, e.g. `str[3]`

- To determine if a substring exists within a string (and where it starts if it does exist), you can use `str.indexOf()`:

  ```
  "decathlon".indexOf("cat")  // Returns 2.
  "team".indexOf("i")  // Returns -1.  There is no
                          // "i" in "team".
  ```

- To access a particular substring, you can use `str.slice()`. There are two ways you can use this:
  - `str.slice(begin, end)` – extracts the characters starting with index `begin` and ending before index `end`.
  - `str.slice(begin)` – extracts the characters starting with index `begin` through the end of the string.

- To transform a string to all lower case, use `str.toLowerCase()`.

- To transform a string to all upper case, use `str.toUpperCase()`.

- To replace all instances of a substring with another string, use `str.replace()`, e.g.:

```
"I love dogs".replace('dog', 'cat')
```

Arrays

- JS has an Array type for representing collections of items. You can declare an array like this:

  ```
  var arr = [ 1, 2, 3 ];
  ```

- Just like with strings, you can access the length of an array with the `length` property. Using `length`, a loop over the elements of an array would look like this:

  ```
  for (var i = 0; i < arr.length; i++) {
       console.log(arr[i]);
  }
  ```

- You can also iterate through an array using the `forEach` method, which takes a function as its argument and calls that function once for every element in the array, passing the array element and the index of that element as arguments. We'll talk more about defining functions later, but here's what it might look like to use `forEach`:

  ```
  function printArrayElem(elem, idx) {
       console.log("The element at index", idx, "is:", elem);
  }

  arr.forEach(printArrayElem);
  ```

- There are several useful functions for converting between arrays and strings:
  - `str.split()` – this function takes a given string and splits it at every occurrence of a specified substring. The split tokens are returned in an Array:

    ```
    "1,2,3,4".split(",")  // Returns [ 1, 2, 3, 4 ]
    ```

  - `arr.join()` – this function takes a given array and joins the elements (as strings) with a specified delimiter string into a single string. You can think of it as the reverse of `str.split()`:

```
[ 1, 2, 3, 4 ].join(",")  // Returns "1,2,3,4"
```

- You can add things to and remove things from the back of an Array using `arr.push()` and `arr.pop()`:

```
var arr = [];
arr.push(1)  // arr = [ 1 ]
arr.push(2)  // arr = [ 1, 2 ]
arr.push(3)  // arr = [ 1, 2, 3 ]
arr.pop()    // Returns 3, arr = [ 1, 2 ]
arr.pop()    // Returns 2, arr = [ 1 ]
```

- You can add things to and remove things from the front of an array using `arr.unshift()` and `arr.shift()`:

```
var arr = [];
arr.unshift(1)  // arr = [ 1 ]
arr.unshift(2)  // arr = [ 2, 1 ]
arr.unshift(3)  // arr = [ 3, 2, 1 ]
arr.shift()     // Returns 3, arr = [ 2, 1 ]
arr.shift()     // Returns 2, arr = [ 1 ]
```

Functions

- A basic function declaration in JS looks something like this:

```
function foo(a, b, c) {
     return a + b + c;
}
```

  - Note that no type is specified for the arguments. We can take advantage of that if we want. For example, these are two valid uses of the same function `foo()`:

```
foo(1, 2, 3);  // Returns 6.
foo("1", "2", "3");  // Returns "123".
```

- You can also leave the arguments unspecified and use the `arguments` keyword to access them as an Array. This allows you to work with a variable number of arguments:

```
function add() {
     var sum = arguments[0];
     for (var i = 1; i < arguments.length; i++) {
          sum += arguments[i];
     }
     return sum;
}
```

- A function can be declared anonymously, for example in an assignment to a variable or even as an argument to another function:

```
var foo = function (a, b, c) {
     ...
};
...
foo(1, 2, 3);
```

OR

```
bar("a", "b", function (err) {
     if (err) {
          // Handle error.
     }
});
```

  - This last example is a common pattern in JS called a ***callback***, where a function is provided to handle the result of some other (possibly long-running) function asynchronously.

- In JS, we often take advantage of variable scoping to use variables with higher scope than the function we're in, e.g.:

```
var data = null;
doHTTPGet(url, function (resp, err) {
        if (err) {
            console.log("Error at URL", url, err);
            return;
        }
        data = resp;
});
```

    - Here, we set `data` inside the callback of `doHTTPGet()`, even though it has higher scope.
    - We also use `url` in the callback to report an error if one occurred.

- ***Closures*** are a special form of this pattern, where we use scoping to create a "private" or "static" variable within a function, e.g.:

```
function makeCounter() {
    var counter = 0;
    return function () {
        return counter++;
    };
}

var count = makeCounter();
count();  // Returns 0
count();  // Returns 1
count();  // Returns 2
```

    - This works because, in JS, the *context* in which a function is run is set *when that function is defined*. In other words, the values of variables from a higher scope that are available within a given function are set when that function is defined.

Objects

- Objects are an incredibly important kind of data in JS. Here's an example JS Object:

```
var person = {
    firstName: "Luke",
    lastName: "Skywalker",
        getFullName: function () {
             return this.firstName + " " +
                  this.lastName;
        }
};
```

- A JS object is specifically a map/dictionary/hash table. Specifically, it maps a set of ***keys*** (the strings `firstName`, `lastName`, and `getFullName` in the example above), to a set of corresponding values. In the example above, the Object stored in `person` contains the following mappings:
  - The key `firstName` maps to the string `"Luke"`.
  - The key `lastName` maps to the string `"Skywalker"`.
  - The key `getFullName` maps to a function that returns a string.

- The values of a JS Object can be any data type, even functions or other Objects. Object keys are always strings.

- Each key/value pair is known as a ***property***. We can access the property values of an object via their keys using either dot or bracket notation:

```
person.firstName
person['firstName']
```

- Bracket notation is particularly useful when the key of the property you want to access is stored in a variable, e.g.:

```
var prop = 'lastName';
console.log(person[prop]);
```

- As demonstrated in the Object above, the `this` keyword can be used to access an object instance from within its property functions.
  - You must call the method on an instance using dot (`.`) or bracket (`[]`) notation to be able to access that instance using `this`, e.g.

    ```
    person.getFullName();
    ```

- In JS, classes are created by writing an Object constructor function and using the ***prototype*** of that constructor function to assign methods and properties to the class (JS is called a prototype-based language).

- For example, here's how we might make a reusable class representing a person:

```
function Person(firstName, lastName) {
     this.firstName = firstName;
     this.lastName = lastName;
}
```

- Here, the `Person()` function is the constructor for the `Person` class. We would create a new `Person` instance using the `new` keyword:

```
var p = new Person("Luke", "Skywalker");
```

  - Under the hood, this creates a new, empty Object and then calls the `Person()` constructor function with `this` set to that newly-created empty Object.
    - Inside the `Person()` constructor function, `this` is used to initialize the properties of the person.

- To assign a class method to the `Person` class, we would add e method to the class's `prototype` object:

```
Person.prototype.getFullName = function () {
     return this.firstName + " " + this.lastName;
};
```

- Here, `Person.prototype` is an Object that is shared by all instances of `Person`.
  - The prototype forms part of a lookup chain (called the ***prototype chain***) that's used to find properties of an object.
  - Here, if you try to access a property of `Person` that isn't set, JS will check `Person.prototype` to see if the property exists there.
  - Thus, `Person.prototype` is available to all instances of `Person` via the `this` keyword, as well.

- We can call methods from the class's prototype like this:

```
p.getFullName();
```