Introduction to JavaScript:

- In 1995, the browser that dominated the market was Netscape Navigator. However, with only static HTML pages, they were eager to improve the experience. They decided they needed a scripting language that would let developers make the internet more dynamic. A Netscape employee named Brendan Eich made the first version in 10 days. After a few iterations, they named the language JavaScript since Java was popular back then. Otherwise the languages don't have much connection.
- Soon, **JavaScript (JS)** was becoming popular. A standard was created by ECMA for scripting languages. The standard was based on JS. **ECMAScript (ES)** is a scripting-language specification standardized by Ecma International. It was created to standardize JavaScript to help foster multiple independent implementations. ES is the standard, while JS implements that standard.
  Note: JavaScript's official name is ECMAScript.
- As time went on, ES's standards improved.
- ES3 (1999) is the baseline for modern day Javascript.
- A bunch of others, like Mozilla, started to work hard on ES5, which was released in 2009.
- Although more versions of the standard have been released, we will mostly talk about new features up to ES6 (2015).
- Browsers adopt new ES standards slowly, but ES6 is mostly completely adopted by modern browsers.
- JavaScript is the programming language of HTML and the Web.
- While HTML is used to define the content of web pages and CSS is used to specify the layout of web pages, JavaScript is used to program the behavior of web pages.
- Javascript is an interpreted programming language that is used to make webpages interactive.
- It's object based.
- It runs on the client's browser.
- It uses events and actions to make webpages interactive.
- JavaScript is a loosely typed and dynamic language. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned and re-assigned values of all types.
- JavaScript is also functional. This means it has **first-class functions**. A programming language is said to have first-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.
- **Note:** Javascript has absolutely nothing to do with Java. It was called JavaScript because at the time it was created, Java was extremely popular.
  "Java is to Javascript is what a car is to a carpet"

Basic JavaScript:

- **Comments:**
- Single line comments are denoted by **//**.
- Multi-line comments are denoted by **/* */**.
- **Linking JavaScript in HTML:**
- There are 3 ways to link JavaScript in HTML:
    1. **Inline:**
    - Here, JavaScript is written in the HTML page but without using the <script> tag.
    - E.g. **<button onclick="console.log("Hello World!);">Click me</button>**
    2. **Embedded:**
    - Here, JavaScript is written in the HTML page in a <script> tag.
    - E.g.

```
<script type="text/javascript">
        console.log("Hello World!);
</script>
```

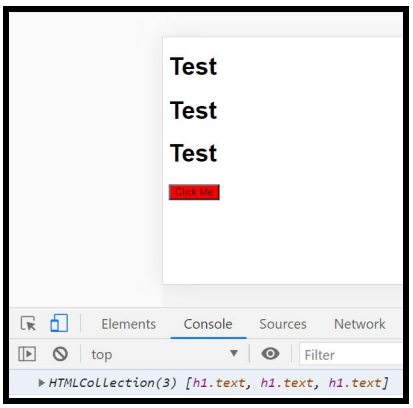
    3. **External:**
    - Here, JavaScript is written in an external .js file and linked to a HTML page using the <script> tag.
    - E.g. **<script src="js/script.js"></script>**
- **Output:**
- JavaScript can output data in different ways:
    1. Writing into an HTML element, using **innerHTML**.
    2. Writing into the HTML output using **document.write()**.
    3. Writing into a pop-up box, using **window.alert()** or **confirm()** or **prompt()**.
    4. Writing into the browser console, using **console.log()**.
- Using innerHTML:
- To access an HTML element, JavaScript can use the **document.getElementById(id)** method. This is the most common way of obtaining an element to modify. Since IDs are unique, we can access a pre-existing element in HTML using this strategy.
- The id attribute defines the HTML element. The innerHTML property defines the HTML content.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            button.style.backgroundColor = "red";
            document.getElementById("text").innerHTML = "Test";
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```

Originally, the page displays “Sample Text”, but after you click on the button, it changes to “Test” because I modified the value of id=“demo” using **document.getElementById("text").innerHTML**.

- **Note:** There is a similar command called **document.getElementsByClassName()**. However since classes are not unique, this command will always return the result in an array. Hence, to change all elements with the given class, you need to loop through the array.
- E.g. Consider the code and output below:

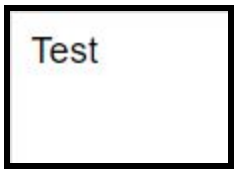
```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            button.style.backgroundColor = "red";
            let x=document.getElementsByClassName("text");  // Find the elements
            console.log(x);
            for(var i = 0; i < x.length; i++){
                x[i].innerText="Test";    // Change the content
            }
        }
    </script>
</head>

<body>

    <h1 class="text"> Sample Text 1 </h1>
    <h1 class="text"> Sample Text 2 </h1>
    <h1 class="text"> Sample Text 3 </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Notice that **x=document.getElementsByClassName("text");** gets you an array. I.e. x is an array.
You can see that x is an array from the bottom right picture on the console. Furthermore, I had to iterate through x to change the elements.

- Using document.write():
- document.write() lets you write onto HTML output.
- **Note:** Using document.write() after an HTML document is loaded, will delete all existing HTML.
- E.g. Consider the code and output below:

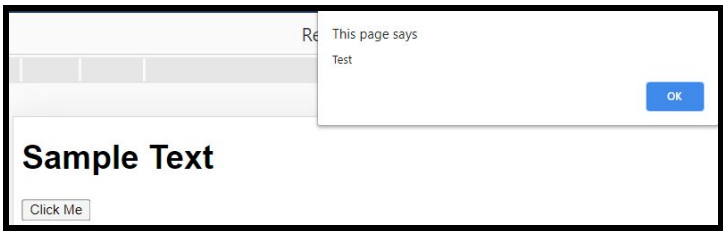
```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            document.write("Test");
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Notice how the HTML output was replaced with “Test” after clicking the button.

- Using window.alert(), prompt() and confirm():
- You can use alert() to create an alert box to display data.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            // button.style.backgroundColor = "red";
            alert("Test");
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```



Notice that after clicking on the button, the pop-up (alert box) appears.

- E.g. For confirm()

```
<script type="text/javascript">
    confirm("Are you sure?");
</script>
```

Confirm() creates a dialog box with “ok” and “cancel” buttons.

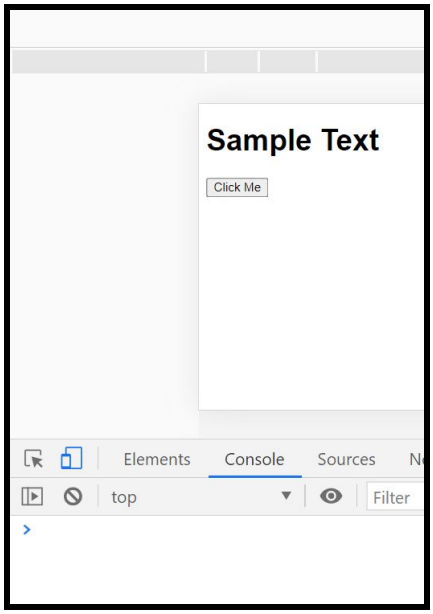
- E.g. For prompt()

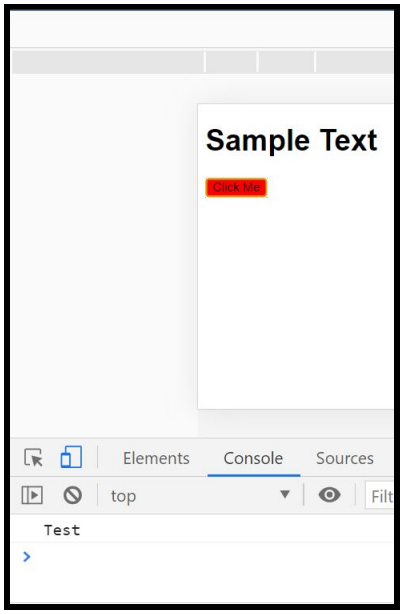
```
<script type="text/javascript">
    prompt("Name?","John");
</script>
```

The prompt() method displays a dialog box that prompts the visitor for input.
A prompt box is often used if you want the user to input a value before entering a page.
You can give a default value. In the example above, the default value is John.

- Using console.log():
- For debugging purposes, you can use the console.log() method to print stuff in the console.
- E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        function changeText(){
            button.style.backgroundColor = "red";
            console.log("Test");
        }
    </script>
</head>

<body>

    <h1 id="text"> Sample Text </h1>
    <!-- Click on the button will invoked the JS function above. -->
    <button id="button" onclick=changeText()> Click Me </button>

</body>
</html>
```





Originally, there's nothing in the console log, but when you click on the button you'll see "Test" there.

- **Scoping:**
- **Scope** determines the visibility or accessibility of a variable or other resource in the area of your code.
- If a variable is declared outside a function, it has **global scope**. This means that all functions can access it.
- Variables declared inside the functions become local to the function and are considered in the corresponding **local scope**. Every function has its own scope. The same variable

can be used in different functions because they are bound to their respective functions and are not mutually visible. Local scope can be divided into **function scope** and **block scope**. The concept of block scope was introduced in ECMAscript 6 (ES6) together with const and let.

- Whenever you declare a variable in a function, the variable is visible only within the function. This is **function scope**. You can't access it outside the function. var has function scope.
- If a variable is declared inside an if statement, switch statement, for or while loop, it has **block scope**. It can only be accessed inside that block.
- **Lexical scope** means the children scope has access to the variables defined in the parent scope.
  E.g. If you have a nested function, the inner function can still access the variables declared in the outer function.
  E.g. If you have an if statement, switch statement, for or while loop in a function, the statement or loop can still access the variables outside it but still within the function.
- **Constants:**
- To declare constants in JavaScript, we use the **const** keyword.
- The value of a constant can't be changed through reassignment, and it can't be redeclared.
- **Note:** The const declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. For instance, in the case where the content is an object, this means the object's contents, its properties, can be altered.
- An initializer for a constant is required. You must specify its value in the same statement in which it's declared. This makes sense, given that it can't be changed later.
- Const has block scope.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    const x = "Hello";
    console.log(x);
    x = "Hi"; // Will give error
</script>
```

```
Hello
▶Uncaught TypeError: Assignment to constant variable.
    at test.html:8
```

Notice that when I try to reassign x, I get an error.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    const x = [1,2,3,4,5,6];
    console.log(x);
    x[0] = 0; // Will not give error
    console.log(x);
</script>
```

```
▶ (6) [1, 2, 3, 4, 5, 6]
▶ (6) [0, 2, 3, 4, 5, 6]
```

In this case, because x is an array, we can change its contents.
However, if you try to change x to an empty array, you'll get an error.

```
<script type="text/javascript">
    const x = [1,2,3,4,5,6];
    console.log(x);
    x = []; // Will give an error.
</script>
```

```
▶ (6) [1, 2, 3, 4, 5, 6]
▶ Uncaught TypeError: Assignment to constant variable.
    at test.html:8
```

- **Variables:**
- In JS, we can declare variables using the **var** or the **let** keyword.
- A variable declared without a value will have the value undefined.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x;
    console.log(x);
</script>
```

```
undefined
>
```

```
<script type="text/javascript">
    var x;
    console.log(x);
</script>
```

```
undefined
>
```

- Variables declared using var have **function scope** while variables declared with let have **block scope**. For this reason, it is better to use let than var.
- **Note:** Originally, var was the only way to declare variables. ECMA 6, which was introduced in 2015, introduced const and let.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    if (1 < 3){
        var x = 5;
    }
    console.log(x);
</script>
```

```
5
>
```

Notice that even though x was created in the if statement, we could still access it outside.

```
<script type="text/javascript">
    if (1 < 3){
        let x = 5;
    }
    console.log(x);
</script>
```

```
Uncaught ReferenceError: x is not defined
    at test.html:9
```

Notice that when we use let, we can no longer access x outside of the if statement. This is because let has block scope while var has function scope.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    console.log(x);
    if (1 < 3){
        var x = 5;
    }
</script>
```

```
undefined
>
```

Notice that even though I didn't declare the variable x, console.log(x) still prints undefined. However, if we change the var to let, shown below, it throws an error. This is due to hoisting.

```
<script type="text/javascript">
    console.log(x);
    if (1 < 3){
        let x = 5;
    }
</script>
```

```
Uncaught ReferenceError: x is not defined
    at test.html:6
```

- **Note:** Even though it's better practice to use let, some old browsers can only handle var.

- **Hoisting:**
- **Hoisting** is JavaScript's default behavior of storing function and variable declarations into memory first.
- **Note:** JavaScript only hoists declarations, not initializations.
- When you run your JavaScript code, the function and variable declarations are put into memory first. Specifically, the variable and function declarations are put into memory during the compile phase.
- **Note:** Functions are hoisted before variables.
- Conceptually, you can think about hoisting as JavaScript's behavior of moving declarations to the top of its scope.
- Since variables declared with var have function scope, if we call a variable declared using var before initializing it, we get undefined.
- E.g. Consider the code and output below:

```
function f1(){
    console.log(a)
    var a = 3
}

f1()
```

```
undefined
```

What's happening is this:

```
function f1(){
    var a // Declaring "a" at the top of the function.
    console.log(a) //While a has been declared, it hasn't been defined, so its value is undefined
    a = 3 // Definition stays in place.
}
f1()
```

- However, if you try to call a variable/constant declared with let or const, respectively, we get an error. This is because let and const have block scope and are initialized at runtime. Furthermore, let and const variables/constants cannot be read/written until they have been fully initialized. Accessing the variable before the initialization results in a ReferenceError. The variable is said to be in a temporal dead zone from the start of the block until the initialization has completed.
- E.g. Consider the code below:

```
{ // TDZ starts at beginning of scope
  console.log(bar); // undefined
  console.log(foo); // ReferenceError
  var bar = 1;
  let foo = 2; // End of TDZ (for foo)
}
```

- **Use Strict:**
- The keyword **"use strict"** defines that JavaScript code should be executed in "strict mode".
- It helps you to write cleaner code, such as preventing you from using undeclared variables.
- Strict mode is declared by adding **"use strict"** to the beginning of a script or a function.
- If it is declared at the beginning of a script, it has global scope. If it is declared inside a function, it has local scope.

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    "use strict"
    y = 4;
    f1();
    function f1(){
        x = 3.14;
    }
</script>
```

```
Uncaught ReferenceError: y is not defined
    at test.html:7
```

Here, since we used “use strict” at the top of the script, it has global scope.

```
<script type="text/javascript">
    y = 4;
    f1();
    function f1(){
        "use strict"
        x = 3.14;
    }
</script>
```

```
Uncaught ReferenceError: x is not defined
    at f1 (test.html:10)
    at test.html:7
```

Here, since we used “use strict” in f1, it has local scope and as a result, y = 4 isn’t affected.

- Strict mode makes it easier to write better, cleaner and more secure JavaScript.
- Strict mode changes previously accepted bad syntax into real errors.
- **Arrays:**
- Can be declared in multiple ways:
    1. **let myArray = new Array();**
       myArray[0] = "JavaScript";
       myArray[1] = "is";
       myArray[2] = "fun";
    2. **let myArray = new Array ("Javascript","is","fun");**
    3. **let myArray = ["Javascript","is","fun"];**
- Array indexes are 0 based, meaning that array[0] is the first element of the array.
- **Data and Structure Types:**
- Because JavaScript is a loosely typed and dynamic language, it has dynamic data types. This means that the same variable can be used to hold different data types. Furthermore, JavaScript does not need you to specify the type of variables.
- The latest ECMAScript standard defines nine data and structure types.
- You can use the **typeof** operator to get the type of any variable.

- A **primitive data value** is a single simple data value with no additional properties and methods.
- There are six data types that are primitives:
    1. **Undefined:**
    - In JavaScript, a variable without a value has the value **undefined**. The type is also undefined.
    - E.g. **let car; // Value is undefined, type is undefined**
    - Any variable can be emptied, by setting the value to undefined. The type will also be undefined.
    - E.g.
      let person = 123;
      person = undefined; // Now both value and type is undefined
    - typeof instance == “undefined”
    2. **Boolean:**
    - Booleans can only have two values: true or false.
    - typeof instance == “boolean”
    3. **String:**
    - typeof instance == “string”
    4. **Number:**
    - typeof instance == “number”
    5. **BigInt:**
    - In JavaScript, BigInt is a numeric data type that can represent integers in the arbitrary precision format.
    - typeof instance == “number”
    6. **Symbol:**
    - A value having the data type Symbol can be referred to as a "Symbol value". In a JavaScript runtime environment, a symbol value is created by invoking the function Symbol, which dynamically produces an anonymous, unique value. A symbol may be used as an object property.
    - typeof instance == “symbol”
- There are 2 structural types:
    1. **Object:**
    - JavaScript objects are written with curly braces {}.
    - JavaScript objects can be declared in multiple ways:
        1. **let myDict = new Object();**
           myDict["first"] = "JavaScript";
           myDict["second"] = "is";
           myDict["third"] = "fun";
        2. **let myDict = {};**
           myDict.first = "JavaScript";
           myDict.second = "is";
           myDict.third = "fun";
        3. **let myDict = {first: "Javascript", second: "is", third: "fun"}**
    - **Note:** The typeof operator returns object for objects, arrays, null, etc.
    - An object in JavaScript is simply a set of key-value pairs.
    - Keys are called **properties**. They must be strings.
    - You can access object properties in two ways:
    1. **objectName.propertyName**
    2. **objectName[“propertyName”]**

    - Properties can be added and changed.
      Note: If you create an object using const, you can still change and/or add properties. You just can't reassign the object.
    - Objects can also have **methods**. Methods are actions that can be performed on objects. A method is a function stored as a property. You access an object method with the following syntax: **objectName.methodName()**.
    - In JavaScript, almost everything is an object.
        - Booleans can be objects if defined with the new keyword.
        - Numbers can be objects if defined with the new keyword.
        - Strings can be objects if defined with the new keyword.
        - Arrays are always objects.
        - Objects are always objects.

    2. **Functions:**
    - typeof instance == "function"

- There is 1 structural root primitive type:

    1. **Null:**
    - In JavaScript null is nothing. It is supposed to be something that doesn't exist.
    - In JavaScript, the data type of null is an object.
    - You can empty an object by setting it to null.
    - E.g.
      ```
      let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
      person = null; // Now value is null, but type is still an object
      ```
    - Undefined and null are equal in value but different in type.
    - typeof instance == "object"

- **If Statements:**
- Syntax:
  ```
  if (condition){
   // Code inside the if block.
  }
  else if (condition){
   // Code inside the else if block.
  }
  else {
   // Code inside the else block.
  }
  ```
- You must have an if block, you can have as many else if blocks as you want, the minimum number of else if blocks is 0, and you can have at most 1 else block, the minimum number of else blocks you can have is 0.
- The conditions are evaluated from top to bottom, and once the first condition is true, the code inside that block will run and once it's done running, it will exit the if-statement.
- **For loops:**
- Syntax:
  ```
  for (statement 1; statement 2; statement 3){
   // Code to run inside loop
  }
  ```
- E.g.
  ```
  for (let i = 0; i < 5; i++){
     console.log(i);
  }
  ```

- **For In Loops:**
- The JavaScript for-in statement loops through the properties of an object that are keyed by strings.
- Syntax:

```
for (variable in object){
 // Code to run inside loop
}
```

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = {'a':1, 'b':2}; // A JavaScript object.
    for (const element in x){
        console.log(element);
    }
</script>
```

```
a
b
>
```

- **For/Of Loop:**
- The JavaScript for/of statement loops through the values of an iterable objects such as Arrays, Strings, Maps, NodeLists, and more.
- **Note:** A JavaScript object is not iterable.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = {'a':1, 'b':2}; // A JavaScript object.
    for (const element of x){
        console.log(element);
    }
</script>
```

```
▶Uncaught TypeError: x is not iterable
    at test.html:7
```

- Syntax:

```
for (variable of iterable) {
  // Code to run inside loop
}
```

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = [1, 2, 3, 4, 5];
    for (const element of x){
        console.log(element);
    }
</script>
```

```
1
2
3
4
5
>
```

- **While loops:**
- Syntax:

```
while (condition) {
  // Code to run inside loop
}
```

- E.g.

```
let i = 1;
while (i < 10){
    console.log(i);
    i++;
}
```

- **Do/While Loop:**
- The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Syntax:

```
do {
  // Code to run inside loop
}
while (condition);
```

- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let i = 1;
    do{
        console.log(i);
        i++;
    }
    while (i < 0)
</script>
```

```
1
>
```

- **Note:** Because the condition is at the end, the loop will always run at least once.
- **Functions:**
- A **function** is a block of code designed to perform a particular task.
- A function is executed when something invokes it.
- Syntax:

```
function function_name(parameter1, parameter2, …, parametern){
 // Code inside the function
}
```

- To invoke a function, put the name of the function, followed by parentheses () and with all the parameters inside the ().
  Note: Accessing a function without the () will return the function object instead of the return value.

- E.g. Consider the code and output below:

```
"use strict"
function f1(a, b, c){
    return(a+b+c)
}
console.log(f1(1, 2, 3)) // Prints the return value of f1.
console.log(f1) // Prints the function object.
```

```
6
ƒ f1(a, b, c){
    return(a+b+c)
}
```

- Because functions in JavaScript are first-class functions, they can be passed around without names. These are known as **anonymous functions** or **lambda functions**.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = function (a, b,){
        return a + b;
    }
    console.log(x(1, 2));
</script>
```

```
3
>
```

- Anonymous functions will be very useful for object methods and callback methods.
- With ES6, **arrow functions** were introduced to allow us to write shorter function syntax.
- E.g. Consider the code and output below:

```
<script type="text/javascript">
    let x = (a, b) => {return a+b;}
    console.log(x(1, 2));
</script>
```

```
3
>
```

This does the same thing as the function in the previous example above, but is much shorter to write.

- Beyond anonymous functions and arrow functions, a function can be passed as an argument to other functions, and can be returned by another function.

- **Equality:**
- There are two main ways of checking for equality in Javascript. We can use either the == operator or the === operator. However, the == operator will sometimes produce odd results. The === operator checks for type equality as well as content equality. A strict comparison (===) is only true if the operands are of the same type and the contents match. However, (==) converts the operands to the same type before making the comparison. Therefore, it is recommended that you only use the === operator.
- E.g.

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

- **Table of JavaScript Arithmetic Operators:**

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- **Table of JavaScript Assignment Operators:**

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

- **Table of JavaScript Comparison Operators:**

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

- **Table of JavaScript Logical Operators:**

Operator	Description
&&	logical and
||	logical or
!	logical not

- **Table of JavaScript Type Operators:**

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Object Oriented Programing in JavaScript:

- **This:**
- The JavaScript keyword **this** refers to the object it belongs to.
- It has different values depending on where it is used:

Item	**What this refers to**
In the global execution context I.e. Outside of any function.	Refers to the global variable. In most browsers, the global variable is Window. **Note: this** will refer to the global object whether in strict mode or not.
In a method I.e. A method is a function in an object.	Refers to the owner object.
In a function without “Use Strict”	Refers to the global variable. In most browsers, the global variable is Window.
In a function with “Use Strict”	Refers to undefined.
In a constructor function I.e. When a function is invoked with the **new** keyword, then the function is known as a **constructor function** and returns a new instance.	Refers to the newly created instance.
In an event	Refers to the element that received the event.

- E.g.
  Here, we have a standalone **console.log(this);** outside of any function. The value of **this** refers to the global object, which is Window.

```
console.log(this);
```

```
▶ Window {window: Window, self: Window, document: document, name: "", location: Location, …}
```

Now, if we use “use strict”, we get the same result.

```
"use strict"
console.log(this);
```

```
▶ Window {window: Window, self: Window, document: document, name: "", location: Location, …}
```

If we use **this** in a method, **this** refers to the owner object.

```
let x = {
    name: "Rick",
    f1: function(){
        return this;
    }
}

console.log(x.f1())
```

```
▶ {name: "Rick", f1: f}
```

Here, the owner object is x, so, **this** represents the object x.

If we use **this** in a function without "use strict", it will refer to the global variable.

```
(function f1(){
    console.log(this);
})();
```

```
▶ Window {window: Window, self: Window, document: document, name: "Test", location: Location, …}
```

If we use **this** in a function with "use strict", it will refer to undefined.

```
(function f1(){
    "use strict";
    console.log(this);
})();
```

```
undefined
>
```

- **New:**
- The **new** operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a **constructor function**.
- 4 things that **new** does (in order):
    1. Creates a new empty object.
    2. Sets the new object's delegate prototype (the object's **__proto__**) to the constructor's prototype property value.
    3. Calls the constructor function with this set to the new object.
    4. Returns the new object or **this**, if an object isn't returned.
- **Constructor Function:**
- A **constructor function** is a function that creates a new object.
- It is often used with the **new** operator.
- E.g.

```
function person(name, age){
    this.name = name;
    this.age = age;
}

let Rick = new person("Rick", 21);
console.log(Rick);
```

```
▶ person {name: "Rick", age: 21}
```

Here, person is a constructor function. We're creating a new person object with the **new** operator.

- **Note:** You cannot add a new property to an object constructor the same way you add a new property to an existing object. To add a new property to a constructor, you must add it to the constructor function.
- E.g.

```
function person(name, age){
    this.name = name;
    this.age = age;
}

person.height = 12; // Won't do anything.

let Rick = new person("Rick", 21);
console.log(Rick);
```

```
▶ person {name: "Rick", age: 21}
```

Even though I wrote **person.height = 12;**, it didn't add the property to the object constructor.

- **__Proto__ and Prototype:**
- When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property, **__proto__**, which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this **prototype chain**.
- **__proto__** is the property of an object that points to the object's delegate prototype.
- **prototype** is the property of a function that points to the object which, when that function is called as a constructor, will be assigned to a new object's __proto__.
- E.g. Consider the code and output below:

```
function person(name, age){
    this.name = name;
    this.age = age;
}

let Rick = new person("Rick", 21);
console.log(Rick.__proto__);
console.log(person.prototype);
console.log(Rick.__proto__ === person.prototype);
```

```
▶ {constructor: ƒ}
▶ {constructor: ƒ}
true
>
```

You can see that Rick.__proto__ and person.prototype points to the same item.

- E.g. Consider the code and output below:

```
function person(name, age){
    this.name = name;
    this.age = age;
}

person.height = 12; // Won't do anything.

let Rick = new person("Rick", 21);
let x = new person("x", 20);
console.log(Rick.__proto__ === x.__proto__);
```

```
true
>
```

  Here, Rick and x are both instances of person, and they have the same __proto__ value.
- All JavaScript objects inherit properties and methods from a prototype:
  - Date objects inherit from Date.prototype.
  - Array objects inherit from Array.prototype.
  - Person objects inherit from Person.prototype.
  - The Object.prototype is on the top of the prototype inheritance chain.
  - Date objects, Array objects, and Person objects inherit from Object.prototype.
- Furthermore, the JavaScript prototype property allows you to add new properties and methods to object constructors.
- E.g. Suppose we have the function Person shown below:

```
function Person(dob){
        this.dob = dob
};
```

  Person.prototype is a property of Person that is created internally once you declare the above function. Many properties can be added to the Person.prototype which are shared by Person instances created using new Person().

  Furthermore, every instance created using **new Person()** has a __proto__ property which points to Person.prototype. This is the chain that is used to traverse to find a property of a particular object.

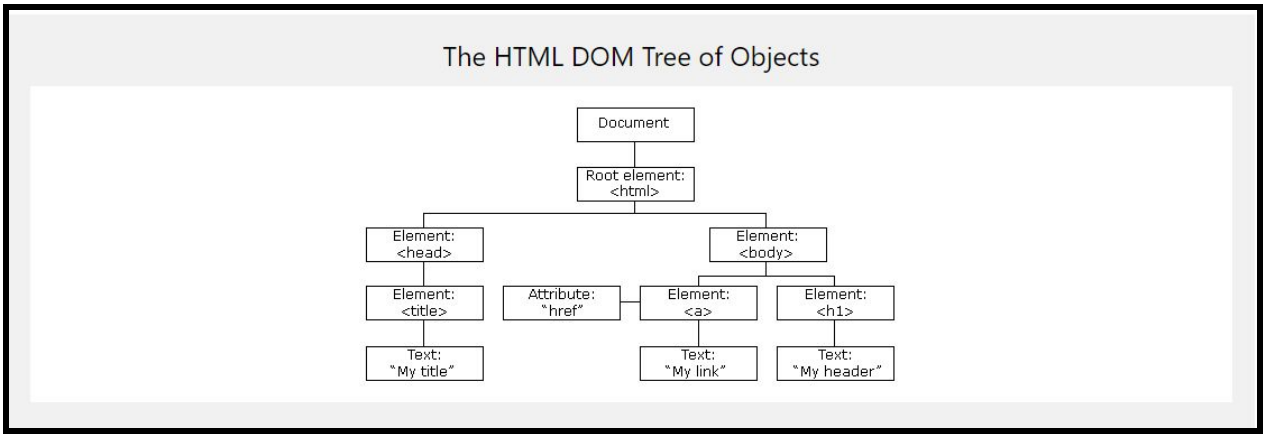
JavaScript in the Browser:

- Javascript in the browser is restrictive.
- You can:
  - Access elements of the webpage and the browser.
  - Track user actions on the webpage (events).
  - Create threads (web workers).
  - Open sockets (web sockets).
- You cannot:
  - Access the file system. However, if you have an upload form, you can still access files uploaded.
  - Access to other programs.
  - Access to other tabs in the browser.

- **The Browser:**
- JavaScript can get you information about the user's browser, such as:

`screen`	the visitor's screen
`browser`	the browser itself
`window`	the current browser window
`url`	the current url
`history`	Back and forward URLs

- **DOM:**
- Stands for **Document Object Model**.
- When a webpage is loaded, the browser creates a DOM of the page.



- The entire document is a document node.
- Every HTML element is an element node.
- The text inside HTML elements are text nodes.
- Every HTML attribute is an attribute node.
- The root node is document.
- To access a node, you can use one of the following ways:
  - **document.getElementById("id");**
  - **document.getElementByTagName("p");**
  - **document.getElementByClassName("class");**
  - **document.querySelector("#id .class p");**
  - **document.querySelectorAll("#id .class p");**

- Table of DOM Methods:

Method	**What it does**
x.innerHTML	Gets the content of x
x.attributes	Gets the attribute nodes of x
x.style	Gets the CSS of x
x.parentNode	Gets the parent node of x
x.children	Gets the child/children nodes of x
x.appendChild	Inserts a child node to x
x.removeChild	Removes a child node from x

Note: When you use these DOM methods, you are manipulating/changing the DOM. For example, when you do x.style, you are manipulating the CSS of the DOM element x, but you are not making any changes to the css file itself.

- The nodes in the node tree have a hierarchical relationship to each other.
- The terms **parent**, **child**, and **sibling** are used to describe the relationships.
- In a node tree, the top node is called the **root** or **root node**.
- Every node has exactly one **parent**, except the root which has no parent.
- A node can have a number of children.
- **Siblings** are nodes with the same parent.
- You can use the following methods to navigate between nodes with JavaScript:
    - parentElement
    - children
    - firstElementChild
    - lastElementChild
    - nextElementSibling
    - previousElementSibling
- **Events:**
- An HTML **event** can be something the browser does, or something a user does.
- All events occur on HTML elements in the browser.
- JavaScript is used to define the action that needs to be taken when an event occurs. I.e. When [HTML Event] , do [JS Action]
- Here are some examples of HTML events:
    - An HTML web page has finished loading.
    - An HTML input field was changed.
    - An HTML button was clicked.

- Table of Basic HTML events:

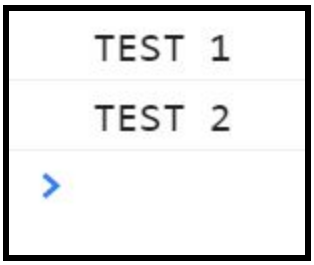
Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page
onsubmit	When a form was submitted.

- An **event listener** in JS programmatically sets an event attribute on an HTML element.
  Syntax: **element.addEventListener(event, functionToExecuteWhenEventOccurs)**
  Note: functionToExecuteWhenEventOccurs is called a **callback function**.
- A **callback** is a function that is designated to be 'called back' at an appropriate time. In the case of events, it will be 'called back' when the event occurs. It can be an anonymous function, or a function defined outside of the event listener.
- There are 2 ways to call a function using addEventListener:
  1. **button.addEventListener('click', function() { alert('Clicked') });**
  2. **function alertClick() { alert('Clicked') }**
     button.addEventListener('click', alertClick);

  In the first way, you're putting the function inside addEventListener and in the second way, you have an external function that you're calling from the addEventListener.
- Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>

</head>

<body>
    <button id="button"> Click Me </button>

    <script type="text/javascript">
        document.getElementById("button").onclick = function(){
            console.log("TEST 1");
        }

        document.getElementById("button").onclick = function(){
            console.log("TEST 2");
        }
    </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>

<head>

</head>

<body>
    <button id="button"> Click Me </button>

    <script type="text/javascript">
        document.getElementById("button").addEventListener("click", function(){
            console.log("TEST 1");
        });

        document.getElementById("button").addEventListener("click", function(){
            console.log("TEST 2");
        });
    </script>
</body>
</html>
```



Notice that in the first set of pictures, when I didn't use addEventListener, only the last function would be invoked. However, when I use addEventListener, both functions are invoked.

- All events that occur create a JS Object with information about that event.
  Event.target gives information about the event origin element.
  Event.type gives information about the type of event.
- Events can be passed to the callback function as an argument.
  E.g.

```
function myCallback(e) {
        // find information about e.
        // execute proper code.
}
```

- We can use JavaScript to create events by using the dispatchEvent function.
- To use the dispatchEvent function, we have to create an Event first.
  We can do it in 2 ways:
  1. `let event = new Event(...);`
     `target.dispatchEvent(event);`
  2. `target.dispatchEvent(new Event(...));`
- E.g. Consider the code below:

```
<!DOCTYPE html>
<html>

<head>

</head>

<body>
    <button id="button" onclick="console.log('Clicked')"> Click Me </button>

    <script type="text/javascript">
        let event = new Event("click");
        button.dispatchEvent(event);

    </script>
</body>
</html>
```

Here, I am using the first way. Each time I refresh the page, I'll see Clicked in the console.

```
<!DOCTYPE html>
<html>

<head>

</head>

<body>
    <button id="button" onclick="console.log('Clicked')"> Click Me </button>

    <script type="text/javascript">
        button.dispatchEvent(new Event("click"));

    </script>
</body>
</html>
```

Here, I'm using the second way. It does the same thing as above.

- addEventListener listens for an event while dispatchEvent creates an event.

Recipes to become a good front-end developer:

- Load Javascript code efficiently.
- Ensure the DOM is loaded with window.onload.

  Note: Suppose you either link an external JavaScript file in the head or you have embedded JavaScript before the <body> tag. Because the HTML file is loaded from top to bottom, if your JavaScript code needs to use something from the HTML code, like id or class, and you don't use window.onload, you'll get an error or an unexpected value. This is because the JavaScript code will get loaded before the HTML code, so the browser can't find the element(s) associated with the id or class.

  E.g. Consider the code and output below:

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        let x = document.getElementById("button");
        console.log(x);
    </script>
</head>

<body>
    <button id="button"> Click Me </button>

</body>
</html>
```

```
null
>
```

Here, I'm trying to get the element with the id "button", which we have. However, because the JavaScript code is loaded before the HTML code, the browser can't find the element with the id "button" as it has not been created yet. Hence, console.log(x) gets us null.

```
<!DOCTYPE html>
<html>

<head>
    <script type="text/javascript">
        window.onload = function(){
            let x = document.getElementById("button");
            console.log(x);
        }
    </script>
</head>

<body>
    <button id="button"> Click Me </button>

</body>
</html>
```

Now, if we use window.onload, the browser will load the HTML code before loading the JavaScript code. As a result, we can get the element with the id “button” now.

- Write good Javascript code.
- Encapsulate Javascript in closures.
- Create a Frontend API

The problem with Javascript interpreters:

- If you write good JavaScript code, it’ll be interpreted by different browsers in the same way. However, if you write bad or sloppy JavaScript code, it’ll be interpreted by different browsers in different ways. This is extremely bad.

How to write good JavaScript code:

1. Use “use strict”

- It forces the browser to validate Javascript against the standard and will dynamically raise errors or warnings in the console when the code is not compliant with the standard.
- See above on page 10.

2. Use JSHint

- Analyze Javascript source code with JSHint.
- JSHint will statically find bugs and report them in the terminal.
- To download JSHint, do **npm install -g jshint**.
- To run JSHint, do **jshint name_of_your_javaScript_file.js**.

The problem with scoping:

- In the browser, all Javascript files share the same execution environment.
  I.e They share the same scope.
  E.g. Consider the code and output below:

```
<script src="test1.js"></script>
<script src="test2.js"></script>
```

```
// Test1.js
let i = 0;
```

```
// Test2.js
console.log(i);
```

```
0
>
```

  Notice that even though i was declared and initialized in Test1.js, we can still call it using Test2.js.
- Because of this, we may get variable and function naming conflicts and strict mode may be applied to all files.
  Imagine you're working at a large company and both you and your colleague decide to use the same name for a variable or function. That could cause a lot of confusion and bugs.
- To fix this, we will encapsulate Javascript in a closure.
- **Closure** is a feature in JavaScript where an inner function has access to the outer function's variables. It allows function/block scopes to be preserved even after they finish executing.
- A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.
- E.g. Consider the code and output below:

```
function f1(){
    let i = 1;
    function f2(){
        return i;
    }
    return f2;
}

x = f1();
console.log(x());
```

```
1
```

  Here I have defined a function within a function. The inner function gains access to all the outer function's local variables, including "i". The variable "i" is in scope for the inner function.
  Normally when a function exits, all its local variables are blown away. However, if we return the inner function and assign it to a variable "x" so that it persists after outer has exited, all of the variables that were in scope when inner was defined also persist.
  Note that the variable "i" is totally private to "x". This is a way of creating private variables in a functional programming language such as JavaScript.
  In a language without closure, the variable "i" would have been garbage collected and thrown away when the function outer exited. Calling "x" would have thrown an error because "i" no longer exists.

- To prevent scoping issues, we can “encapsulate” our JavaScript code in a function, like such:

```
(function() {
        "use strict";
        let private = function() {
                // private is not available from outside
        }
}());
```

- We can emulate private variables and methods using closure.
- E.g. Consider the code below:

```
let counter = (function() {
  let privateCounter = 0;

  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
      changeBy(-1);
    },

    returnValue: function() {
      return privateCounter;
    }
  };

})();

console.log(counter.returnValue());  // 0.

counter.increment();
counter.increment();
console.log(counter.returnValue());  // 2.

counter.decrement();
console.log(counter.returnValue());  // 1.
```

Here, the variable privateCounter and the function changeBy are private. You can’t access them outside of the anonymous function. However, you can access the 3 public functions, increment, decrement and returnValue from outside the anonymous function.

- Every closure has three scopes:
    1. Local Scope
    2. Outer Functions Scope
    3. Global Scope
- **Note:** When you do something like

```
(function(){
…
})();
```

This is called an **IIFE (Immediately Invoked Function Expression)**.
It is a JavaScript function that runs as soon as it’s defined.

HTML5:

- HTML5 has a lot of new features such as:

1. **Geolocation:**

- The HTML Geolocation API is used to get the geographical position of a user.
- Since this can compromise privacy, the position is not available unless the user approves it.
- The **getCurrentPosition()** method is used to return the user's position.
- E.g.

```
navigator.geolocation.getCurrentPosition(success);
function success(position){
        let lat = position.coords.latitude;
        let long = position.coords.longitude;
}
```

2. **Local Storage:**

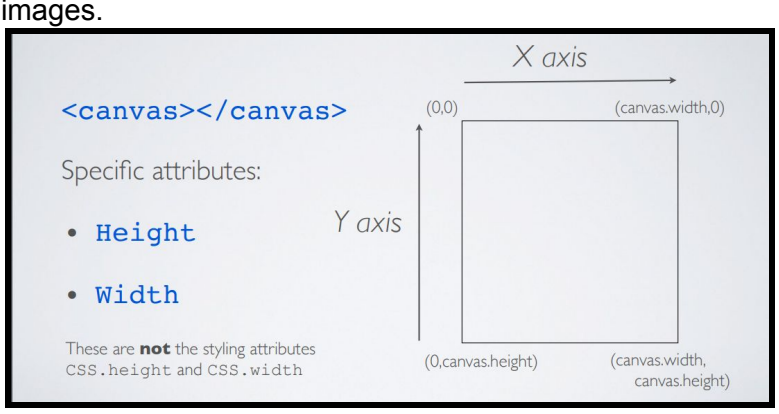
- The localStorage object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year.
- **Note:** Local storage is not cookies.
- Local storage stores key/value pairs.
- The storage is bound to the origin (domain/protocol). That is, different protocols or subdomains infer different storage objects, they can't access data from each other.
  I.e. Accessible from the same domain only.
- On Chrome, we can store up to 10 mb in local storage.

3. **Drag and Drop:**

- Can be used for interacting with the DOM or uploading a file.

4. **Canvas:**

- The HTML <canvas> element is used to draw graphics via JavaScript.
- The <canvas> element is only a container for graphics. You must use JavaScript to actually draw the graphics.
- Canvas has several methods for drawing paths, boxes, circles, text, and adding images.



5. **Video:**

- The HTML <video> element is used to show a video on a web page.
- You can embed Youtube videos too.

6. **Speech to Text:**
7. **Text to Speech:**