JavaScript Introduction

JavaScript

JavaScript is the most commonly used **client-side** scripting language. It is a **high-level, multi-paradigm, interpreted** programming language used to create dynamic webpages. When the browser loads a webpage, the browser interprets JavaScript code in it and executes it.

When we say JavaScript is a client-side language, we mean that it runs in the user's web browser and not on a server. However, although JavaScript originated as a scripting language that runs in the browser, the Node.js runtime environment does allow JavaScript code to be run on servers as the backend program for an application.

When we say JavaScript is a high level language, we mean that it abstracts away many implementation details that relate to computer hardware - like allocating memory or garbage collection of objects. When we say it is multi-paradigm, which means it supports many programming paradigms like procedural, object-oriented, and functional programming.

ECMAScript is the official name for JavaScript. It is used for language specification. The versions of a JavaScript are defined by ECMAScript (ES). The versions of JavaScript are ES5, ES6, ES7, and so on.

In HTML, JavaScript code is written inside the `<script>` and `</script>` tags. You can place any number of scripts in an HTML document. Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page, or in both.

```
<script>
 JavaScript code
</script>
```

Internal JavaScript - When the JavaScript code placed anywhere within the HTML page using `<script>` tags, then it called Internal JavaScript.

External JavaScript - When the same JavaScript code used for many webpages then we place the JavaScript code in a separate file. When JavaScript code placed in a separate file from the HTML code is called External JavaScript. These files are saved with the ". js" extension and imported into the HTML page using the `src` attribute. The src attribute specifies the URL/path of an external JavaScript file. (E.g.: `<script src="./script.js"></script>`)

Syntax

JavaScript is case sensitive. Every statement in JavaScript is separated using a semicolon (;). JavaScript ignores multiple spaces and tabs.

JavaScript Comments: JavaScript support single-line as well as multi-line comments. Single line comment starts with `//` and multi-line comments are wrapped between `/*` and `*/`.

JavaScript Literals: Literals are the fixed values, it can be numbers, strings, boolean values, etc. The number type stores integer, float, and hexadecimal value. Strings are text, enclosed within single or double quotes (' or "). If the number is enclosed with single or double quotes, then it is considered as a string.

JavaScript Keywords: Keywords are tokens that have special meaning in JavaScript. The keywords supported by JavaScript are break, case, catch, continue, do, else, finally, for, function, if, in, new, return, switch, this, throw, try, typeof, var, void, while, etc.

JavaScript Variables

Variables are used to store data values. It uses the `var` keyword to declare variables. An equal sign (=) is used to assign values to variables. Variable names are identifiers that should not be a JavaScript keyword. It starts only with the alphabet, underscores (_) or dollar ($). It cannot start with a number and also there shouldn't be spaces in-between.

Example:

```
<script>
    var x, y ; //declaring the variable
    x = 6; y = 6; // assigning values
    var name = 'John';
</script>
```

Operators

JavaScript Operators performs some operation on single or multiple operands and produces a result. The categories of operators and the operators used in JavaScript are listed below:

- Arithmetic operators - `+, -, *, /, %, ++, --`
- Comparison Operators - `==, ===, !=, >, <, >=, <=`
- Logical Operators - `&&, ||, !`
- Assignment Operators - `=, +=, -=, *=, /=, %=`
- Ternary Operator - `<condition> ? <value1> : <value2>;`

Control Flow

JavaScript uses the following control flow statements:

- `if/else if/else`
- `for`
- `for-in`
- `for-of`
- `while`
- `do-while`

The important difference between `for-in` and `for-of` is that the first allows iteration over the **keys** of any object, while `for-of` allows iteration over an array or array-like object. The syntax is demonstrated below.

```
let person = {
  name: 'Bob',
  age: 25
};

for (let prop in person) {
  console.log(person[prop]); // prints 'Bob' and then 25
}

let people = [
{
  name: 'Alice',
  age: 30
},
{
  name: 'Charlie',
  age: 29
}
]

for (let pers of people) {
  console.log(pers.name); // prints 'Alice' then 'Charlie'
}
```

Compared to Other Languages (Java/C#)

When comparing to other languages, it's interesting to note that JavaScript was originally developed to be a scripting language with similar syntax to Java. While this means that some of the syntax bears a resemblance to the C-derived languages of Java and C#, JavaScript is very different from them in more meaningful ways.

- As a dynamically-typed language, JavaScript does not declare types and a variable you had previously stored a string in might hold a number. This also means that that there is no static type checking available, leading to potential issues where a function can return a value of multiple types. In Java or C#, this is far less likely as

you usually declare a specific type for each variable, and static type checking is a feature leading to less confusion in utilizing an API.
- JavaScript is single-threaded and runs off of an event-loop. Java and C# support multi-threading.
- JavaScript can run on a server or on a browser. Java and C# would have to be compiled to WebAssembly in order to run on modern-day browsers.
- JavaScript is a multi-paradigm language, meaning we can solve our problems in a variety of ways, functionally, object-oriented, or event-driven. With Java and C#, while many of those paradigms have support in the language, Object-oriented solutions are preferred. In JavaScript, functions are first-class variables, meaning that they are treated like any other variable and can be passed as arguments to other functions.
- JavaScript utilizes prototype-based objects. Java and C# utilize class-based objects. In ES6, JavaScript introduced class-based syntax which allows us to interface with our prototypes in a similar manner to Java and C#. However, OOP in JavaScript is accomplished very differently than in these other languages. For example:
    1. An "overridden" method is merely shadowed on the prototype and is therefore still accessible.
    2. JavaScript has no concept of overloading.
    3. Encapsulation is possible in JavaScript but there are no access modifiers and is not as simple as in other languages.

Data types

JavaScript includes the 7 following data types. The primitive data types used in JavaScript are:

- string
- number
- boolean
- null
- undefined
- object
- Symbol

Symbol is a new data type introduced in ES6 but is not commonly used.

Also, in this section we'll discuss about Arrays and Date specifically, which are both Objects.

Loosely- Typed JavaScript - JavaScript is a dynamic or loosely-typed language because a variable can hold the value of any data type at any point in time.

Example:

```
var a = 100;

a = true;
```

```
a = null;
a = undefined;
a = "Johnson";


alert(a); // Johnson
```

NOTE: The `alert()` method displays an alert box in the current browser window with a specified message and an 'OK' button.

Data types supported by JavaScript are listed below:

Strings

Strings are text, enclosed within a single(') or double quotes ("). We can have quotes inside the string if they don't match the enclosing quotes.

Example:

```
var name  = "Johnson";
var a ='hello';
var a = "Let's have a cup of coffee."; // single quote inside double quotes
var b = 'He said "Hello" and left.';  // double quotes inside single quotes
var c = 'We\'ll never give up.';     // escaping single quote with backslash
```

Numbers

Numbers can be positive or negative numbers, with or without decimals and exponential representation of numbers. If a number is enclosed within single or double quotes, then it is considered a string. The Number data type also includes special values such as Infinity, NaN (Not-a-Number value). When we divide a number by 0, it results in infinity. When we do undefined mathematical operations such as taking the square root of -1, multiplying, or dividing with String the result is NaN. Example:

```
var a = 51;         // integer
var b = 15.5;       // floating-point number
var c = 1.5e-4;    // exponential notation, equivalent to 0.0001.5
var d = 1.5e+4;    // exponential notation, equivalent to 1.5e4 or 15000
var h = -6/0 // results infinity
var j = ("2" / 2); //results NaN
```

BigInt

In JavaScript, the “number” type cannot represent integer values larger than 253 or less than -253 for negatives. BigInt type used to represent integers of arbitrary length. A BigInt is created by appending n to the end of an integer literal.

Example:

```
 var bigInt = 1234567890123456789012345678901234567890n; //// the "n" at the end refe
rs it's a BigInt
```

Boolean

In JavaScript, we have two Boolean values: true and false.

Example:

```
var isEmpty = true;
var isGreater = (5>6); //returns false
(5 == 5) //returns true;
```

The “null” value

A null value refers to nothing and is not equivalent to an empty string ("") or 0.

Example:

```
var a = null;
alert(a); // Output: null
```

The “undefined” value

The undefined refers to value is not initialized. If a variable declared but not initialized with a value, its value is undefined.

Example:

```
let x;
alert(x); // shows "undefined"


var car = "";    // It is an empty string not undefined.
```

NOTE: An empty string has legal value and type defined.

Objects

An object contains a set of key-value pairs. A key is a name that is of string type. The value can be any data type such as numbers, booleans, strings, function, arrays, and other objects.

The syntax of a JavaScript object as follows: `var <object-name> = { key1: value1, key2: value2,... key: valueN};`

Example:

```
var person1 = {
    firstName: "John",
    lastName: "Wilson",
    age: 23,
 };
```

You can access the values of an object's properties using dot notation (.) or bracket ([ ]).

Example:

```
person1.firstName; // returns John
person1.lastName; // returns Wilson
or
person1["firstName"];// returns John
person1["lastName"];// returns Wilson
```

Object Constructor - Another way to create an object is with Object Constructor using the `new` keyword. You can attach properties and methods to the object using dot (`.`) or bracket (`[]`).

Example:

```
var person2 = new Object();
// Attach properties  to person object
person2.firstName = "Ben";
person2["lastName"] = "Tennyson";
person2.age = 18;
```

If we try to access properties or call methods that don't exist, then JavaScript will return `undefined`. The `hasOwnProperty()` method is used to check whether an object has a particular property or not.

Example:

```
var person3 = new Object();
person3.firstName; // returns undefined
if(person3.hasOwnProperty("firstName")){
            person3.firstName;
    }
```

We can use `for..in` loop to get the list of all properties and methods of an object.

Example:

```
for(var key in person1){
        alert(key);
    };
```

Arrays

Arrays stores a list of values under a single variable.

The Syntax for Arrays in JavaScript as follows. `var <array-name> = [element0, element1, element2,... elementN];`

Array values are associated with an index starts with 0. The array includes "length" property that returns the number of values stored in it.

Example:

```
var arr = [1,  2.2,  "hello",  false];
alert(arr[0]);  //Outputs 1
alert(arr[1]);  //Outputs 2.2
alert(arr[2]); //Outputs  hello
alert(arr[3]);  //Outputs false
alert(arr.length); //Outputs 4
```

Arrays are also dynamic - we can add elements to grow the array and remove elements to shrink it. Below are some common array methods:

- `.push()`
- `.pop()`
- `.filter()`
- `.map()`
- `.slice()`

Array Constructor - We can also initialize an array using a `new` keyword.

Example:

```
var names = new Array();
names[0] = "John";
names[1] = "Ben";
names[2] = "Chris";
```

```
var numbers =  new Array (1, 2, "three", 4);
```

Date

JavaScript provides a Date object to work with date & time. It includes days, months, years, hours, minutes, seconds, and milliseconds in each object.

We can display the current date and time by creating a Date object using Date() or new Date().

Example:

```
Date(); //current date


var currentDate = new Date(); //current date
```

We can create a date object by specifying different parameters in the Date constructor:

```
var dt = new Date();
var dt = new Date(milliseconds);
var dt = new Date('date string');
var dt = new Date(year, month[, date, hour, minute, second, millisecond]);
```

Example for creating Date objects:

```
var date1 = new Date("3 march 2015");
var date2 = new Date("3 February, 2015");
var date3 = new Date("3rd February, 2015"); // invalid date
var date4 = new Date("2015 3 February");
```

Typeof Operator

The typeof operator returns the data type of its operand.

Syntax: `typeof operand` or `typeof (operand)`

Example:

```
typeof 42; // output: "number"


typeof NaN ; // output: "number"


typeof 'hello'; // output: "string"
```

```
typeof true; // output: "boolean"


var a;
typeof a; // output: "undefined"


typeof 42n; // output: "bigint"


function func(){}
typeof func; // output: "function"


// For arrays, date, regular expression,null and objects , typeof operators returns "
object"
typeof {a: 1}; // output: "object"


typeof [1, 2, 4]; // output: "object"


typeof new Date(); // output: "object"


typeof null;    // output:  "object"
```

Variable Scope:

The Variable scope defines the lifetime and visibility of a variable. Each variable associated with a scope. The variable can be accessed only within its scope.

Global Scope

Variables defined outside any function, block, or module have global scope. The global scope variables are accessed everywhere in the application.

Example:

```
var a = 'Hello World!'; // This is a global variable
function greeting() {
    console.log(a);
}
greeting(); // Outputs 'Hello World!'
```

The global variable's lifetime is throughout the application.

Example:

```
var app = {}; // A global object
app.foo = 'Homer';
app.bar = 'Marge';
function func() {
    console.log(app.foo);
}
func(); // Outputs 'Marge'
```

We can declare the global variables in a browser using the global `window` object.

Example:

```
window.app= { value: 1 };
window.b = "Hello!!!"
```

Local Scope

Local Scope used to refer to Function-local scope, but following the introduction of block scope, this term is considered ambiguous and should not be used. The local scope has divided into the function scope, the block scope and the lexical scope. The block scope concept is introduced in ECMA Script 6 (ES6) together with the new ways to declare variables - `const` and `let`.

Function Scope

The variable declared in a function is only visible inside that function. `var` is the keyword to define variable for a function-scope accessibility. These variables cannot be accessed or modified.

```
//global scope
function func1(){
    //function scope 1
    function func2(){
        //function scope 2
    }
}

//global scope
```

```
function func3(){
    //function scope 3
}

//global scope
```

Example:

```
function func(){
    var animal ='Lion'; //exist in function scope
    console.log('inside function: ',animal);
}

func();                    //Output: "inside function: Lion"
console.log(animal);       //error: animal is not defined
```

Block Scope

Block scope is the scope of the variables declared inside the {} (curly brackets). In ES6, `const` and `let` keywords allow developers to declare variables in the block scope, which means those variables exist only within the corresponding block.

Example:

```
function func(){
    if(true){
        var fruit1 = 'apple';        //exist in function scope
        const fruit2 = 'banana';     //exist in block scope
        let fruit3 = 'strawberry';   //exist in block scope

    }
    console.log(fruit1);
    console.log(fruit2);    // results error - due to it exist in block scope
    console.log(fruit3);    // results error - due to it exist in block scope
}

foo();
```

The result will be:

```
apple
error: fruit2 is not defined
error: fruit3 is not defined
```

Lexical Scope

Lexical scope is that a variable defined outside a function can access the inside another function defined after the variable declaration. The inner functions are lexically bound to the execution context of their outer functions.

Example:

```
function func1(){
    var animal1 = "Lion";   //exist in Lexical scope

    function func2(){    //Inner Function

        var animal2 = "Tiger"; //exist in function scope
        console.log(animal1);
        console.log(animal2);

    }
    func2();
    console.log(animal2); // results error - due to it exist in function scope
}

foo1();
```

The result will be:

```
Lion
Tiger
error: animal2 is not defined
```

Hoisting

In JavaScript, variable declarations made with `var` and function declarations made with the `function` keyword are **hoisted** - or moved - to the top of the scope in which they are declared when the JavaScript interpreter parses the code. This means that variables and functions can be used **before they are even declared** as shown below.

```
function example() {
  // var a declaration hoisted here
  a = 4;
  var a;
  a++;
  console.log(a); // prints 5
}
// anotherExample declaration hoisted to here
anotherExample(); // no error thrown!

function anotherExample() {
  console.log('it works!');
}
```

Arrays

An array is a variable that allows the programmer to store more than one value. Arrays in JavaScript are objects and thus consist of key/value pairs and inherit from the Array prototype. Like objects, array values can consist of JavaScript primitives, or other JavaScript objects, including arrays and functions.

Creating an Array

In JavaScript, arrays can be created using square brackets, using what is known as an array literal. They can also be created using the `new` keyword, but it is best practice to use array literals.

```
// array literal
let cheeses = ['bleu', 'cheddar', 'parmesan', 'brie']
// ["bleu", "cheddar", "parmesan", "brie"]
// new keyword
let primes = new Array(2, 3, 5, 7, 11, 13)
// [2, 3, 5, 7, 11, 13]
```

Both methods create an array object based on the Array Prototype.

Array Structure

All array objects share a common structure. Each array has a `length` field that stores the current length of the array. In addition, the prototype of an array is `[]`, giving each array access to certain functions that we will cover later.

```
let array = [1, 2, 3]
console.log(array.length) // 3
console.log(array.__proto__) // []
```

Accessing an Array

Arrays in JavaScript are **zero-indexed**, meaning that the first element in an array is represented by the key `0`. In order to access our array, we can use Array Notation:

```
let cheeses = ['bleu', 'cheddar', 'parmesan', 'brie']
console.log(cheeses[0]) // bleu
```

Here, we gave the zero-index to obtain the first element in the array, which is the string `bleu`. The string `cheddar` would be represented by the key `1`, `parmesan` would be `2`, and `brie` would be `3`. The length of the array is `4`. The last key of an array is normally `array.length - 1`, however, be aware that there might not be data at this location.

```
//access the last element of the array
console.log(cheeses[cheeses.length-1])
```

Modifying an Array

You can also assign a different value by using the index and an assignment operator.

```
cheeses[2] = 'american' // changes parmesan to american
```

Changing the Array Length - JavaScript is a dynamic language and arrays are no different. the length of an array can be changed in several ways.

Adding an Item - To add an item to an array you can specify an index

```
// create an empty array
let arr = []
console.log(arr.length) // 0
// add an element at a specific index
arr[0] = 'duck'
console.log(arr.length) // 1
console.log(arr) // ["duck"]

arr[3] = 'chicken'
console.log(arr.length) // 4
```

```
console.log(arr) // [ "duck", <2 empty slots>, "chicken"]
```

In the above example, we first added the string `duck` to an empty array, changing the length of that array from 1 to 0. Note however, that this can be dangerous, as we can specify any index we wish. We then added the string `chicken` at index 3, changing the length of our array to 4, with two empty indexes. One way around this would be to use the `length` field to determine what the next index should be.

```
arr[arr.length] = 'pigeon'
console.log(arr.length) // 5
console.log(arr) // [ "duck", <2 empty slots>, "chicken", "pigeon"]
```

An easier and safer way to add a new element to an array is the `push()` method inherited from the Array Prototype.

```
let students = []
students.push('Timothy')
students.push('Zach')
console.log(students.length) // 2
console.log(students) // ["Timothy", "Zach"]
```

Removing an Item - Removing an item from an array in JavaScript can result in unexpected behavior if done incorrectly. When removing a key from an object, you might use the `delete` keyword. When you do this with an array, the length field will not be updated, resulting in an empty slot.

```
let arr = [1, 2, 3]
// [1, 2, 3]
console.log(arr.length) // 3
delete arr[0]
// [<empty slot>, 2, 3]
console.log(arr.length) // 3
```

In order to remove elements we turn to the methods `splice()`, `pop()`, and `shift` from the Array Prototype. The `splice()` method allows us to remove a number of elements at a specific index. It can also allow us to replace those elements with something else. `splice()` returns an array containing all the values removed from the array.

NOTE: In most browsers, if you do not specify the number of elements to remove, it will remove all elements starting at the index specified.

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
arr.splice(2, 1) // returns [3]
console.log(arr)
```

```
// [1, 2, 4, 5, 6, 7, 8, 9]
arr.splice(4, 3) // returns [6, 7, 8]
console.log(arr)
// [1, 2, 4, 5, 9]
```

The `pop()` method removes and returns the last element in an array and the `shift()` method removes and returns the first element in an array.

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
arr.pop() // returns [9]
console.log(arr)
// [1, 2, 3, 4, 5, 6, 7, 8]
arr.shift() // returns [1]
console.log(arr)
// [2, 3, 4, 5, 6, 7, 8]
```

NOTE: - The `length` field can be modified directly. This can lead to arrays behaving a little oddly. If you change the `length` field to a value less than the actual length of the array, JavaScript will remove elements from the array in most browsers. If you set the field to a greater value, it will "add" empty fields. The fields that are added only exist in the `toString` method, as no keys are created.

Iterating Through an Array

You can use a for loop to iterate through an array in JavaScript:

```
let list = [1, 2, 3, 4, 5];


// standard for
for(let i = 0; i < list.length; i++){
    console.log(list[i]);
}
```

Arrays are iterable, and so you can use a `for-in` or `for-of` loop to iterate through an array. `for-in` will iterate through the keys of an array. `for-of` will iterate through the values of the array.

```
// for-in
for(key in list) {
    // an enhanced for-in loop in JS iterates through
    // the keys not the values
```

```
    console.log(key);
    console.log(list[key]);
}


// for-of
for(element of list) {
    // if you say of, it will iterate through the values
    console.log(element);
}
```

There is also the `forEach()` method on the Array Prototype. This function is a functional array method that takes in a callback function and runs that function for each element in the array. The `forEach()` method returns undefined.

```
// forEach()
list.forEach(
    function(value, index) {
        console.log(index + ' ' + value);
    }
)
```

Array Methods

The Array Prototype includes many useful methods. Above, we covered the `splice()`, `shift()`, `push()`, `pop`, and `forEach()`. Here we will talk about a few other useful methods.

sort()

The `sort()` method will sort an array "in-place". This means that the array will be modified and the original array will be sorted. The `sort()` method can take in a function that will compare objects and sort them to your preference. If no callback function is provided, each element is converted to a string and sorted in ascending order.

```
let list = ['tiger', 'panda', 'giraffe', 'cat', 'owl', 'bird']
list.sort()
// this will sort alphabetically
console.log(list)
// [ "bird", "cat", "giraffe", "owl", "panda", "tiger" ]


// This will sort by size of the string
```

```
list.sort(function(item1, item2) {return item1.length - item2.length})

console.log(list)

// [ "cat", "owl", "bird", "panda", "tiger", "giraffe" ]
```

indexOf() & lastIndexOf()

The `indexOf()` method returns the first index at which an element is present. `lastIndexOf()` returns the last index at which an element is present. If the element can't be found in the list, both will return the value `-1`.

```
let list = [1, 1, 2, 3, 3, 5, 6, 1]

list.indexOf(3) // returns 3

list.lastIndexOf(3) // returns 4

list.indexOf(1) // returns 0

list.lastIndexOf(1) // returns 7

list.indexOf('cat') // returns -1

list.lastIndexOf('cat') // returns -1
```

find() & findIndex()

The `find()` method returns the first element in an array for which the callback function returns a truthy value. The `findIndex()` method does the same but returns that element's index. For example, if I were to try to find the first string in an array that had a length greater than or equal to 7, I could write a function that tested for that and pass it to the `find()` method:

```
let words = ['the', 'small', 'fox', 'ate', 'a', 'largish', 'breakfast', 'and', 'slept
']

words.find(function(word) {return word.length >= 7}) //returns 'largish'

words.findIndex(function(word) {return word.length >= 7}) //returns 5
```

filter()

The `filter()` function takes a callback function and creates a new array that is the made up of elements for which the callback function returns a truthy value. This can be useful in situations where you wish to perform an operation on only a subset of elements in an array. The original array is not modified. For example, I may wish to get only even numbers from an array:

```
let list = [4, 67, 34, 55, 79, 12]

let evens = list.filter(function(n) { return n%2 === 0 })

console.log(evens)

// [ 4, 34, 12 ]
```

map()

The `map()` function takes a callback function and creates a new array that is the result of calling the function on each element of the array. The original array is not modified. For example, if I were to have an array of numbers and I wished to have an array of the squares of those numbers, I could use the `map()` function to accomplish this:

```
let numbers = [1, 2, 3, 4, 5, 6, 7]

let squares = numbers.map(function(n) {return n*n})

console.log(squares)

// [ 1, 4, 9, 16, 25, 36, 49 ]
```

reduce()

The `reduce()` function takes a callback function and returns a single value that is the result of calling the function for each value in the array. For example, if we wished to have the sum of each element in an array, we could do the following:

```
let numbers = [1, 2, 3, 4, 5, 6, 7]

let result = numbers.reduce(function(previousValue, currentValue) {return previousVal
ue + currentValue})

console.log(result) // 28
```

JavaScript this keyword

The `this` keyword is a reference variable that refers to the current object.

- `this` **alone**: refers to a global Object.

Example:

```
var x = this;
```

- `this` **in function**: refers to the Global object [object Window].

Example:

```
function myFunction() {
  return this;
}
```

- `this` **in** `strict` **mode**:

Example:

```
"use strict";
var x = this; //Here, this refers to the Global object [object Window]


"use strict";
function myFunction() {
  return this; //Here,  this is undefined.
}
```

- `this` **in Event Handlers**: refers to the HTML element that received the event

Example:

```
<button onclick="this.style.backgroundColor= 'green'">
    Click Me!
</button>
```

- `this` **in Object Method Binding**: refers to the object. In the below example, `this` refers to people object.

Example:

```
let people = function(name, age) {
         this.name = name;
         this.age = age;


    this.displayInfo = function() {
       document.write(this.name + " is " + this.age + " years old");
      }
    }
```

- **The call() and apply() method**: allows us to write a method that can be used on different objects. Here, person1 object writes its fullName function on person2 object using call() method

Example:

```
var person1 = {
  fullName: function() {
```

```
    return this.firstName + " " + this.lastName;
  }
}
var person2 = {
  firstName:"John",
  lastName: "Wilson",
}
document.write("Hello, "+ person1.fullName.call(person2));
```

strict mode

JavaScript is a loosely typed scripting language. JavaScript allows strictness of code by using `"use strict";` statement at the top of JavaScript code or in a function.

For example, when we expect the compiler to give an error if we have used a variable before defining it, then we apply "strict mode" to the JavaScript code.

```
<!DOCTYPE html>
<html>
<body>
        <h1>strict mode</h1>

        <p id="errorMessage"></p>

        <script>
                "use strict";
                try
                {
                        var x = 16; // valid in strict mode
                        y = 10; // error
                }
                catch(ex)
                {
                        document.getElementById("errorMessage").innerHTML = ex;
                }
```

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

The output will be:

```
strict mode
ReferenceError: y is not defined
```

The strict mode in JavaScript does not allow us to use undefined variables, reserved keywords as variable or function name, duplicate properties of an object, duplicate parameters of the function, assign values to read-only properties, Modifying arguments object, and Deleting an undeletable property.

Example:

```
<script>
try
{
        "use strict";

        x = 1; // error
        var for = 19; // error
        var break = 5; // error

        var person = { name: "John", name: "Ben" }; //error
        function divide(val, val){return val / val }; //error

        var arr = [1 ,2 ,3 ,4, 5];
        arr.length = 10; //error

}
catch(ex)
{
    document.getElementById("errorMessage").innerHTML = ex;
}
</script>
```

Strict mode can be applied to function level in order to implement strictness only in that particular function.

Example:

```
x = 1; //valid


function sum(val1, val2){
    "use strict";


     result = val1 + val2; //error


    return result;
}
```

JSON

JSON (**J**ava**S**cript **O**bject **N**otation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

JSON Object is a set of *key and value pair* enclosed within curly braces. A key is a string enclosed in quotation marks. A value can be a string, number, boolean expression, array, or object. A key value pair follows a specific syntax, with the key followed by a colon followed by the value. Key/value pairs are separated by comma. (`"name" : "Andy"`)

Example:

```
var Book = {
"id": 110,
"language": "Python",
"author": ["John", "Ben"]
};
```

We can store multiple JSON objects in Arrays.

Example:

```
var student =[


     {
        "id":"01",
        "name": "Tom",
        "lastname": "Price"
     },
```

```
    {
       "id":"02",
       "name": "Nick",
       "lastname": "Thameson"
    }
 ] ;
```

Some of the applications of JSON are listed below:

- Used to transmit data between the server and web application using JSON.
- JSON format helps transmit and serialize all types of structured data.
- Allows us to perform asynchronous data calls without the need to do a page refresh.
- Web services and Restful APIs use the JSON format to get public data.

Type Coercion

`==` vs. `===`

`==` is used for comparison between two variables irrespective of the data type of variable. `===` is used for comparison between two variables but this will check strict type, which means it will check data type and compare two values.

Example:

```
5 == '5' // Returns true
5 === '5' // Returns False
```

`==` is a type converting equality that automatically converts the string ('5') to number (5). It does implicit type conversion of variables.

`===` is strict equality also compares the data type of the variable. It returns true only if the data type and value of the two variables are the same.

Type coercion

Type coercion is the process of converting a value from one data type to another data type.

Explicit type coercion - We can explicitly convert the data type of the variable. *For example:* `Number('3')`, `String (123)`, `Boolean(2)`

Implicit type coercion - JavaScript is a loosely-typed language, values can also be converted between different types automatically. It usually happens when you apply operators to values of different types. *For example:* `'3' * '2'`, `2/'5'`, `123 + ''`

String conversion

To explicitly convert values to a string apply the `String()` function. Implicit coercion is triggered by the binary `+` operator when any operand is a string.

Example:

```
String(123) // explicit
123 + ''    // implicit
```

Boolean conversion

To explicitly convert a value to a boolean apply the `Boolean()` function. Implicit conversion happens in logical context, or is triggered by logical operators (`|| && !`).

Example:

```
Boolean(21)          // explicit
if (2) { ... }      // implicit due to logical context
7 || 'hello'        // implicit due to logical operator
```

Number conversion

To explicitly convert a value to a boolean apply the `Number()` function. Implicit coercion is triggered by comparison operators (`>, <, <=,>=`), bitwise operators (`| & ^ ~`), arithmetic operators (`- + * / %`),unary `+` operator and loose equality operator`==`.

Example:

```
Number('123')   // explicit
+'123'          // implicit
123 != '456'    // implicit
```

toString() method - used to convert values to String.

```
(100 + 23).toString() //returns "123"
true.toString()    // returns "true"
```

JavaScript automatically calls the variable's toString() function when you try to "output" an object or a variable.

```
obj = {name:"Ben"}  // toString converts to "[object Object]"
arr = [1,2,3,4]     // toString converts to "1,2,3,4"
date = new Date()   // toString converts to "Fri Jul 18 2014 09:08:55 GMT+0200"
```

Truthy and Falsy in JavaScript

Falsy value

In JavaScript, any expressions or value that results in boolean `false` value, are considered as Falsy. The falsy values/expressions in JavaScript are:

1. Obviously boolean `false` is `false`.
2. Any empty string will be evaluated to `false`.
3. Any `undefined` variable will be equal to `false`.
4. Any `null` variable will be equal to `false`.
5. Any numerical expression with result in `NaN` (not a number) will be equal to `false`.
6. Any numerical expression with result in zero will be equal to `false`.

Truthy value

In JavaScript, any expressions or value that results in boolean `true` value, are considered as Truthy. Any expression or value other than above listed falsy values – is considered truthy.

Example:

```
'Hello'  // truthy
if(1){}          // truthy
if(-1){}         // truthy
new Boolean(false);     // is truthy values because 'object' is always true
new String('')    // is truthy values because 'object' is always true
```

JavaScript Functions

A function is a group of reusable code which can be called anywhere in the program. A JavaScript function is defined using the `function` keyword. The syntax for creating a function:

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

A simple JavaScript Function:

```
function showMessage() {
  alert( 'Hello everyone!' );
```

```
}


//function call
showMessage();  // outputs "Hello everyone!" in alert box.
```

JavaScript can return a value by executing the code after the function call. To return a value, we use the `return` keyword.

```
var product = multiplyFunc(4, 3);   // Function called and the return value is stored in 'product' variable.


function myFunction(a, b) {
  return a * b;              // Function returns the product of a and b
}
```

Simple Snippet: To convert centimeter to feet using JavaScript Functions

```
<!DOCTYPE html>
<html>
<body>


<h2>JavaScript Function for converting centimeter to feet</h2>


<p id="demo"></p>


<script>
function toFeet(cm) {
  return (cm/30.48);
}
document.getElementById("demo").innerHTML = toFeet(180);
</script>


</body>
</html>
```

The output will be:

```
JavaScript Function for converting centimeter to feet
```

```
5.905511811023622
```

Pass by value

In JavaScript, all function arguments are passed by value. This means that the value of any variable passed to a function is copied into the argument of the function. Any changes you make to the argument will not be reflected in the variable outside of the function.

Primitives

With primitive values this behavior is straightforward. The primitive value is copied to a new variable:

```
function changeValue(number) {
  console.log(number) // 20
  number = 42
  console.log(number) // 42
}
let myNumber = 20
changeValue(myNumber)
console.log(myNumber) // 20
```

In the above example, we defined a primitive value `myNumber` to be 20. When we passed this variable into the `changeValue` function, it copied the value 20 into the new `number` variable. When we changed `number` it did not affect `myNumber` because those are two different variables, each with their own value.

Objects

If you pass an object into a function, the story is slightly different. The value that is stored in a variable containing an object is not the object itself. Instead, an object reference is being stored inside of that variable. When you pass a variable containing a reference to an object, that reference is copied into the arguments of the function. Since the new variable has a copy of that object reference, we can use this variable to modify the object.

```
let myObject = {'pet': 'Cat'}
console.log(myObject.pet) // 'Cat'
function adoptDog(obj) {
  obj.pet = 'Dog'
}
adoptDog(myObject)
console.log(myObject.pet) // 'Dog'
```

It is tempting to conclude that objects are pass by reference, because you can modify the object that we pass into the function. However, if we attempt to change the value of the variable by assigning a new object we see that this isn't true:

```
let myObject = {'pet': 'Cat'}
console.log(myObject.pet) // 'Cat'
function adoptDog(obj) {
  obj = {'pet': 'Dog'}
}
adoptDog(myObject)
console.log(myObject.pet) // 'Cat'
```

Here, we see that because we reassigned the variable `obj` to a new object, the value of the variable changed and so the value of the variable `myObject` did not change. JavaScript is pass by value.

Function Expression / Anonymous Function

Function Expressions also are known as a named or anonymous function. An anonymous function is a function declared without any identifier refer to it. It is an expression that the variable holds a function. For example: `var x = function (a, b) {return a * b};`

Example:

```
var anon = function() {
  alert('I am anonymous');
};
var prd = function (a, b) {
    return a * b;

};
anon();
alert("prd = " + prd(2,4));
```

The above example results in two alert boxes on the current browser. The first alert box has "I am anonymous" inside it. The second alert box has "prd = 8" inside it.

Self-Invoking Functions / IIFE Functions

A self-invoking function is an anonymous function that is invoked immediately after its definition. It is also known as the IIFE (Immediately Invoked Function Expression) function. It holds an anonymous function inside a set of parentheses (), which does the execution.

Syntax: `(function(){ code goes here...})();`

Example:

```
(function(){
    // do this right now
    console.log("Look at me, I'm running");
})();
```

Callback Functions

A callback function is a function that gets executed after another function completes the execution. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors. JavaScript runs the code in sequential order (from top to down). If there is a case that code runs after some other execution, which is not happening in a sequence is called **asynchronous programming**. All functions in JavaScript are objects and a JavaScript function can be passed another function as an argument.

A callback function can be created by using the `callback` keyword as the last parameter.

Example for callback functions:

```
function funcOne(x) { alert("x = " + x); }


function funcTwo(y, callback) {
    callback(y);
}


funcTwo(2, funcOne);
```

In the above example, `funcOne` is the callback function. When `funcTwo(2, funcOne);` is called, `funcTwo` takes in a variable (y) and a function (funcOne). `funcTwo` then passes the variable (y=2) to the function it took in, i.e. `funcOne(2)` is called. Then, issues an alert with `x=2` on the current browser.

We can also pass an anonymous functions as a callback function.

Example:

```
function funcTwo(y, callback) {
    callback(y);
    callback(y);
}
```

```
functionTwo(10, function(x) { alert("x = " + x); })
```

The above example issues an alert two times, saying `x = 10` on the current browser.

Closures

A closure is a function that remembers and accesses the variables and arguments of its outer function even after the function return. The closure able to access the variables defined between its curly brackets, the outer function's variables and the global variables.

Example:

```
function greeting() {
    var message = 'Hi';

    function sayHi() {
        console.log(message);
    }

    return sayHi;
}
let hi = greeting();
hi(); // prints "hi" in the console.
```

Normally, when the `greeting()` function has completed executing, the `message` variable is no longer accessible. In this case, we execute the `hi()` function that references the `sayHi()` function, the `message` variable still exists. Hence, the `sayHi()` function is a closure.

Hoisting of Functions and Variables

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Hoisting of Variables:

The JavaScript compiler moves all the declarations of variables to the top so that there will not be any error.

Example:

```
<script>
        //line 1
        x = 1;
```

```
        document.getElementById("p1").innerHTML = x ;
        var x;
</script>
```

NOTE: var x; declaration moved to the top (at line 1) of their scope.
JavaScript Hoisting only moves the variable declaration to the top, not the variables that are declared and initialized in a single line.

Example:

```
alert('x = ' + x); // displays x = undefined


var x = 1;
```

NOTE: Since hoisting is only possible with the declaration but not the initialization, `var x = 1;` not moved to the top of their scope.

Hoisting of Functions:

JavaScript compiler moves the function definition to the top in the same way as a variable declaration.

Example:

```
alert(Sum(5, 5)); // output: 10


function Sum(val1, val2)
{
    return val1 + val2;
}
```

JavaScript compiler moves only the Function declaration to the top, not the function expression.

Example:

```
add(5, 5); // Results an error


var add = function sum(a, b)
{
    return a+b;
}
```

Hoisting Functions Before Variables:

JavaScript compiler moves a function's definition before the variable declaration.

Example:

```
alert(UseMe); // displays the UseMe Function definition


var UseMe = "UseMe Variable";


function UseMe()
{
    alert("UseMe function called");
}
```

The above example will display "UseMe" function definition because JavaScript compiler moves the function before variables. Therefore, the alert box displays `function UseMe() { alert("UseMe function called");}`.

NOTE: The above code doesn't display "UseMe Variable" in the alert box due to hoisting functions before variables.

async and await keyword

Async and Await are extensions of promises.

async -An async function is a function that operates asynchronously via the event loop, returns a Promise object implicitly. `async` ensures that the function returns a `promise`.

await - The `await` keyword is only valid inside `async` functions. `await` makes JavaScript wait until that promise settles and returns its result.

Example: Returns a promise object

```
async function asyncFunc() {
  let response = await fetch(protectedUrl);
  let text = await response.text(); // response body consumed
  document.write(text);
}


asyncFunc();
```

OOP in JS

Encapsulation via Closures

Encapsulation means hiding information or data. The simplest a way to create encapsulation in JavaScript is using closures. A closure can be created as a function with private state. When creating many closures sharing the same private state, we create an object.

Example:

```
const Book = function(t, a) {
   let title = t;
   let author = a;

   return {
      summary : function() {
        console.log(`${title} written by ${author}.`);
      }
   }
}
const book1 = new Book('Hippie', 'Paulo Coelho');
book1.summary(); // Returns Hippie written by Paulo Coelho.
```

Prototypical inheritance

Object Prototypes (`__proto__`)- All JavaScript objects have a prototype. Browsers implement prototypes through the `__proto__` property.

Function prototypes (`prototype`) - In JavaScript, all functions are also objects, which means that they can have properties. Any time you create a function, it will automatically have a property called `prototype`. Thus, Functions also have a `prototype` property.

When we call a function with new, it sets the returned object's `__proto__` property equal to the function's `prototype` property. This is the key to inheritance.

Inheritance in JavaScript is implemented through the **prototype chain**. Every normally created object, array, and function has a prototype chain of `__proto__` properties ending with `Object.prototype` at the top.

Example: Here, we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal". The animal properties and methods are become automatically available in rabbit. Such properties are called “inherited”.

```
let animal = {
  eats: true
walk() {
    alert("Animal walk");
  }
```

```
};
let rabbit = {
  jumps: true
  __proto__: animal      // sets animal to be a prototype of rabbit.
};

// we can find both properties in rabbit now:
alert( rabbit.eats ); // true
alert( rabbit.jumps ); // true
// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

In the above example, when alert tries to read property `rabbit.eats`, it's not in rabbit but JavaScript follows the prototype reference and finds it in animal. Also the `walk()` method is automatically taken from the prototype.