Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

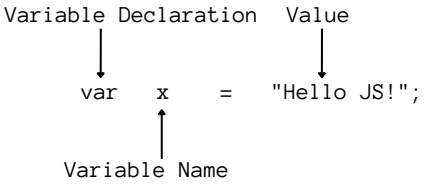
What is Javascript?

- JavaScript is **lightweight**, **interpreted or JIT compiled programming language** with **first-class functions**.
- Most well-known as **the scripting language for web pages**, many **non-browser environments also use it**, such as NodeJs and Apache CouchDB.
- JS is a **prototype-based**, **loosely typed**, **dynamic scripting language**, **supporting object oriented**, **functional programming styles**.

Who invented Javascript?

JavaScript was invented by **Brendan Eich in 1995 within 10 days.**

Javascript variables and data types



Javascript variable Declaration Types

- Non-Block-Scope Variable Declarations do not care about the block scope and able to access anywhere in the file.

  (var)

- Block-Scope Variable Declarations care about the block scopes and are unable to access everywhere file in the file except the block-scope.

  (let, const)

Javascript variable Rules

1. **Use descriptive names**: Choose names that clearly describe the purpose or content of the variable. This makes your code more readable and understandable.
2. **Start with a letter, underscore, or dollar sign: Variable names must begin with a letter, underscore (_), or dollar sign ($). They cannot start with a number.**.
3. **Avoid reserved words: Do not use JavaScript reserved words, such as var, function, if, else, for, etc., as variable names.**
4. **Be case-sensitive**: JavaScript is case-sensitive, so myVariable and myvariable are considered two different variables.
5. **Use meaningful abbreviations**: While it's important to use descriptive names, if a variable name becomes too long or cumbersome, you can use meaningful abbreviations to keep it concise. However, make sure the abbreviation is understandable and does not sacrifice clarity.

Javascript Data types

1. Number
2. String
3. Boolean
4. null
5. undefined
6. Symbol
7. BigInt

Primitive data types

8. Object
9. Array
10. Function

Object data types

Loosely typed and dynamically typed

Loosely typed

In a loosely typed language like JavaScript, variables are not bound to a specific data type. This means that you can assign values of different types to the same variable without explicitly declaring its type. JavaScript allows implicit type coercion, which means it automatically converts values between different types when needed.

```
let num = 10; // variable 'num' is loosely typed
num = "Hello"; // assigning a string value
num = true; // assigning a boolean value
```

Dynamically typed

JavaScript is a dynamically typed language, which means that variable types are determined at runtime. Unlike statically typed languages where variable types are checked during compile-time, JavaScript determines the type of a variable based on the value assigned to it at runtime. This allows for more flexibility as variables can change their types during the execution of a program.

```
let num = 10; // 'x' is dynamically typed as a number
num = "Hello"; // 'x' dynamically changes its type to a string
num = true; // 'x' dynamically changes its type to boolean
```

Console

In JavaScript, the console object provides a set of methods that allow you to interact with the browser's debugging console. You can use these methods to log messages, display information, debug code, and more.

1. **console.log()**: Prints a message or object to the console.
2. **console.error()**: Prints an error message to the console with an error icon.
3. **console.warn()**: Prints a warning message to the console with a warning icon.
4. **console.info()**: Prints an informational message to the console with an info icon.
5. **console.clear()**: Clears the console.
6. **console.table()**: Displays tabular data as a table.

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

JS String Methods

charAt():
Returns the character at a specified index in a string.

```
let str = "Hello, World!";
let char = str.charAt(7); // Returns 'W'
```

charCodeAt():
Returns the Unicode of the character at a specified index in a string.

```
let str = "Hello, World!";
let unicode = str.charCodeAt(0); // Returns 72 (Unicode value for
'H')
```

concat():
Combines two or more strings

```
let str1 = "Hello";
let str2 = "World";
let result = str1.concat(", ", str2); // Returns "Hello, World"
```

endsWith():
Checks whether a string ends with specified characters.

```
let str = "Hello, World!";
let endsWithWorld = str.endsWith("World!"); // Returns true
```

fromCharCode():
Converts Unicode values into characters.

```
let char = String.fromCharCode(72); // Returns 'H'
```

includes():
Checks whether a string contains the specified substring.

```
let str = "Hello, World!";
let includesHello = str.includes("Hello"); // Returns true
```

indexOf():
Returns the index of the first occurrence of a specified value in a string.

```
let str = "Hello, World!";
let index = str.indexOf("World"); // Returns 7
```

lastIndexOf():
Returns the index of the last occurrence of a specified value in a string.

```
let str = "Hello, World, World!";
let lastIndex = str.lastIndexOf("World"); // Returns 13
```

localeCompare():
Compares two strings in the current locale.

```
let str1 = "apple";
let str2 = "banana";
let comparison = str1.localeCompare(str2); // Returns a value
indicating the comparison result
```

match():
Searches a string for a specified value or regular expression pattern and returns an array of the matches.

```
let str = "The quick brown fox jumps over the lazy dog.";
let matches = str.match(/o/g); // Returns an array of 'o' occurrences
```

replace():
Searches a string for a specified value or regular expression pattern and replaces it with another string.

```
let str = "Hello, World!";
let newStr = str.replace("World", "Universe"); // Returns "Hello,
Universe!"
```

search():
Searches a string for a specified value or regular expression pattern and returns the index of the first match.

```
let str = "Hello, World!";
let index = str.search("World"); // Returns 7
```

slice():
Extracts a part of a string and returns a new string.

```
let str = "Hello, World!";
let sliced = str.slice(7, 12); // Returns "World"
```

split():
Splits a string into an array of substrings based on a specified delimiter.

```
let str = "Apple, Banana, Cherry";
let arr = str.split(", "); // Returns an array ["Apple", "Banana",
"Cherry"]
```

startsWith():
Checks whether a string starts with specified characters.

```
let str = "Hello, World!";
let startsWithHello = str.startsWith("Hello"); // Returns true
```

substr() & substring:
Extracts a specified number of characters from a string, starting at a specified index.

```
let str = "Hello, World!";
let substring = str.substr(7); // Returns "World!"
let substring = str.substring(7, 12); // Returns "World"
```

toLocaleLowerCase():
Converts a string to lowercase according to the host's locale.

```
let str = "Hello, World!";
let lowerCase = str.toLocaleLowerCase(); // Returns "hello, world!"
(locale-dependent)
```

toLocaleUpperCase():
Converts a string to uppercase according to the host's locale.

```
let str = "Hello, World!";
let upperCase = str.toLocaleUpperCase(); // Returns "HELLO, WORLD!"
(locale-dependent)
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

toLowerCase():
Converts a string to lowercase.

```
let str = "Hello, World!";
let lowerCase = str.toLowerCase(); // Returns "hello, world!"
```

toUpperCase():
Converts a string to uppercase.

```
let str = "Hello, World!";
let upperCase = str.toUpperCase(); // Returns "HELLO, WORLD!"
```

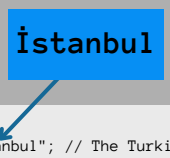
toLocaleLowerCase() / toLocaleUpperCase() and toLowerCase() / toUpperCase() are both JavaScript string methods used to convert the characters in a string to lowercase / uppercase. However, they have a key difference related to localization and handling of special characters.

toLowerCase() / toUpperCase():

- toLowerCase() / toUpperCase() is a basic string method that converts all characters in a string to lowercase / uppercase based on the Unicode character set.
- It performs a simple, one-to-one mapping of characters, making it suitable for most cases where you want to convert text to lowercase / uppercase.

toLocaleLowerCase() / toLocaleUpperCase():

- toLocaleLowerCase() / toLocaleUpperCase() is similar to toLowerCase() in that it converts characters to lowercase / uppercase. However, it also takes into account the current locale (language and region) settings of the user's environment.
- It can be useful when you need to handle text in a way that respects the linguistic rules of a specific locale, especially for languages with special characters or letter casing rules that differ from the default Unicode behavior.




```
let str = "İstanbul"; // The Turkish dotted capital 'I'
let lowerCase = str.toLocaleLowerCase("tr-TR"); // Returns "istanbul"
```

trim():
Removes whitespace from both ends of a string.

```
let str = "   Hello, World!   ";
let trimmed = str.trim(); // Returns "Hello, World!"
```

valueOf():
Returns the primitive value of a string object.

```
let str = new String("Hello, World!");
let primitiveValue = str.valueOf(); // Returns "Hello, World!"
```

---

length:
Get the length of the string

```
let str = "Hello, World!";
str.length; // Returns 13
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

Javascript Operators

Arithmetic Operators

- Addition (+): Adds two values together.
- Subtraction (-): Subtracts one value from another.
- Multiplication (*): Multiplies two values.
- Division (/): Divides one value by another.
- Modulo (%): Returns the remainder of the division.

Assignment Operators

- Assignment (=): Assigns a value to a variable.
- Addition assignment (+=): Adds a value to a variable and assigns the result to the variable.
- Subtraction assignment (-=): Subtracts a value from a variable and assigns the result to the variable.
- Multiplication assignment (*=): Multiplies a variable by a value and assigns the result to the variable.
- Division assignment (/=): Divides a variable by a value and assigns the result to the variable.
- Modulo assignment (%=): Calculates the remainder of the division and assigns the result to the variable.

Comparison Operators

- Equality (==): Checks if two values are equal.
- Inequality (!=): Checks if two values are not equal.
- Strict equality (===): Checks if two values are equal in value and type.
- Strict inequality (!==): Checks if two values are not equal in value or type.
- Greater than (>): Checks if one value is greater than another.
- Less than (<): Checks if one value is less than another.
- Greater than or equal to (>=): Checks if one value is greater than or equal to another.
- Less than or equal to (<=): Checks if one value is less than or equal to another.

Logical Operators

- Logical AND (&&): Returns true if both operands are true.
- Logical OR (||): Returns true if either operand is true.
- Logical NOT (!): Negates a boolean value.

Unary Operators

- Unary plus (+): Converts the operand into a number.
- Unary minus (-): Negates the operand.
- Increment (++): Increases the value of a variable by 1.
- Decrement (--): Decreases the value of a variable by 1.

Ternary Operator

- Ternary operator (condition ? expression1 : expression2): Evaluates the condition and returns expression1 if the condition is true, otherwise returns expression2.

String Operators

- Concatenation (+): Concatenates two strings together.

Control flow (if statements, loops)

Logical AND (&&): Returns true if both operands are true. Logical OR (||): Returns true if either operand is true. Logical NOT (!): Negates a boolean value.

If Statements

The "if" statement allows you to execute a block of code conditionally based on a specified condition.

```
if (condition) {
  // Code to be executed if the condition is true
} else if (condition2) {
  // Code to be executed if condition2 is true
} else {
  // Code to be executed if none of the conditions are true
}
```

Loops

- Loops allow you to repeatedly execute a block of code as long as a condition is true or for a specified number of iterations.
- JavaScript provides different types of loops: "for", "while", and "do-while".

For Loop

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

While Loop

```
let i = 0;
while (i < 5) {
  console.log(i); i++;
}
```

Do-While Loop

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

Javascript Functions

Common Syntax of a function

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

There are several types of functions that you can use based on your requirements and coding style.

Main Functions

Named Functions (*A function as a statement*)

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}
greet("John"); // Output: Hello, John!
```

Function Expressions (A function as an expression)

```
const greet = function(name) {
  console.log(`Hello, ${name}!`);
};
greet("John"); // Output: Hello, John!
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

Arrow Functions

```
let name = (arguements1, arguements2, arguements 3...) => {
statements
};
```

Function constructor

```
var F = new Function(arg1, functionBody)
var F = new Function(arg1, arg2, functionBody)
var F = new Function(arg1, arg2, ..........., argN,
functionBody)
```

Other Functions

Anonymous Function

```
function () {
// function body
}
```

Immediately Invoked Function Expressions

```
(function () {
console.log("Welcome to Javascript");
})();
```

Callback Functions

```
function process(callback) {
  // Do some processing
  callback();
}
function callback() {
  console.log("Callback function called!");
}
process(callback); // Output: Callback function called!
```

Arrays

There are 2 main ways to create an array.

1. Literal Based
2. *Constructor Based*

But, They introduce another 2 ways to define arrays in JS with ES6 in 2015

- Array.from()
- *Array.of()*

☞ Literal Based

```
let array1 = [];
array1 = [1, 2, 3, 4, 5, 6];
```

☞ Constructor Based

```
let array2 = new Array(1, 2, 3, 4, 5, "Software
Developer", function () {alert("IJSE")});
```

Array Methods

push():
Adds one or more elements to the end of an array and returns the new length of the array.

```
let myArray1 = ["Ranil", "Mahindha", "Anura"];
myArray1.push("Ranjan");
console.log(myArray1); // ['Ranil', 'Mahindha', 'Anura', 'Ranjan']
```

pop():
Removes the last element from an array and returns that element.

```
myArray1 = ['Ranil', 'Mahindha', 'Anura', 'Ranjan', 'Karuu'];
myArray1.pop();
console.log(myArray1); // ['Ranil', 'Mahindha', 'Anura', 'Ranjan']
```

shift():
Removes the first element from an array and returns that element.

```
myArray1 = ['Ranil', 'Mahindha', 'Anura'];
myArray1.shift();
console.log(myArray1); // ['Mahindha', 'Anura']
```

unshift():
Adds one or more elements to the beginning of an array and returns the new length of the array.

```
myArray1 = ['Mahindha', 'Anura'];
myArray1.unshift('Sajith'); // return 3 here
console.log(myArray1); // ['Sajith', 'Mahindha', 'Anura']
```

concat():
Joins two or more arrays and returns a new array.

```
let array1 = [1, 2, 3];
let array2 = [4, 5, 6, 7, 8, 9, 10];

array1.concat(array2); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
array2.concat(array1); // [4, 5, 6, 7, 8, 9, 10, 1, 2, 3]
```

slice():
Returns a shallow copy of a portion of an array into a new array.

```
let numberArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

console.log(numberArray.slice(2)) //[3, 4, 5, 6, 7, 8, 9, 10]
console.log(numberArray.slice(4)) // [5, 6, 7, 8, 9, 10]
console.log(numberArray.slice(-1)) // [10]
console.log(numberArray.slice(-2)) // [9, 10]
console.log(numberArray.slice(2, 6)) //[3, 4, 5, 6]
```

splice():
Adds or removes elements from an array at a specified position.

```
let arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

arr1.splice(7);
console.log(arr1); // [1, 2, 3, 4, 5, 6, 7]
arr1.splice(5);
console.log(arr1); // [1, 2, 3, 4, 5]
arr1.splice(1, 3);
console.log(arr1); // [1, 5]
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

indexOf():
Returns the first index at which a given element can be found in the array.

```
let arr2 = [1, 2, 3, 4 ,5];
console.log(arr2.indexOf(3)); // 2
```

lastIndexOf():
Returns the last index at which a given element can be found in the array.

```
arr2 = [1, 2, 3, 4 ,5, 2];
console.log(arr2.lastIndexOf(2)); // 5
```

includes():
Determines whether an array includes a certain value among its entries.

```
arr2 = [1, 2, 3, 4 ,5, 2, "Hello"];
arr2.includes(4); //true
arr2.includes(7); //false
arr2.includes("Hello"); // true
arr2.includes("hello"); // false
```

filter():
Creates a new array with all elements that pass the test implemented by the provided function.

```
arr2 = [1, 2, 3, 4 ,5, 2];
arr2.filter(function(item) {
    return item % 2 === 0
})
// [2, 4, 2]

arr2.filter(function(item) {
    return item % 2 > 0
})
// [1, 3, 5]
```

map():
Creates a new array with the results of calling a provided function on every element in the array.

```
arr2 = [1, 2, 3, 4 ,5, 2];
arr2.map((item, index) => {console.log(index)});
```

reduce():
Applies a function against an accumulator and each element in the array to reduce it to a single value.

```
arr3 = [1, 2, 3, 4];
let result = arr3.reduce((acum, currentvalue) => acum + currentvalue)
console.log(result); // 10
```

forEach():
Executes a provided function once for each array element.

```
arr3 = [1, 2, 3, 4];
arr3.forEach(function(item) {
    console.log(item);
})
```

sort():
Sorts the elements of an array in place and returns the array.

```
let arr7 = [20, 10, 40, 30, 15];
arr7.sort() // [10, 15, 20, 30, 40]
```

reverse():
Reverses the order of the elements in an array.

```
let arr7 = [20, 10, 40, 30, 15];
aarr7.reverse() // [40, 30, 20, 15, 10]
```

some():
Tests whether at least one element in the array passes the test implemented by the provided function.

```
let arr9 = [2, 4, 5];
let y = arr9.some(function(item) {return item % 2 > 0});
console.log(y); // true

let arr9 = [2, 4, 8];
let y = arr9.some(function(item) {return item % 2 > 0});
console.log(y); // false
```

every():
Tests whether all elements in the array pass the test implemented by the provided function.

```
arr9 = [1, 3, 4];
let rs = arr9.every((item) => item % 2 > 0); //false

arr9 = [1, 3, 5];
rs = arr9.every((item) => item % 2 > 0); // true
```

join():
Joins all elements of an array into a string.

```
arr9 = [1, 3, 5];
arr9.join("-") //'1-3-5'
arr9.join(",") //'1,3,5'
arr9.join("*") //'1*3*5'
arr9.join("->") //'1->3->5'
arr9.join(">>") //'1>>3>>5'
```

toString():
Returns a string representing the specified array and its elements.

```
arr9.toString(); // '1,3,5'
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

Objects

In JavaScript, an object is a fundamental data type that represents a collection of key-value pairs, where each key is a string (or Symbol) and each value can be any JavaScript data type, including other objects. **Objects in JavaScript are used to model real-world entities, concepts, or data structures.**

Here are the most common ways to create objects in JavaScript,

1. Constructor Function
2. Class Syntax (Introduced in ECMAScript 2015)
3. Object Literal
4. Object.create()

☞ Constructor Function

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

const person = new Person("John", 30);
```

☞ Class Syntax

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

const person = new Person("John", 30);
```

☞ Object Literal

The simplest way to create an object is using the object literal notation, which involves defining the object properties and values within curly braces {}.

```
const person = {
  name: "John",
  age: 30
};
```

☞ Object.create()

The Object.create() method allows you to create a new object with a specified prototype object.

```
const personPrototype = {
  greet: function() {
    console.log("Hello!");
  }
};

const person = Object.create(personPrototype);
person.name = "John";
person.age = 30;
```

Private Attributes in Object

There is **no built-in mechanism to define private attributes directly.** However, you can use various techniques to simulate private attributes and achieve encapsulation.

☞ Naming Convention

You can prefix the private attribute with an underscore _. Although this doesn't enforce privacy, it indicates to other developers that the attribute is intended to be private and should not be accessed directly.

```
function Person(name, age) {
  this._name = name; // Private attribute
  this.age = age; // Public attribute
}

Person.prototype.getName = function() {
  return this._name;
};

const person = new Person("John", 30);
console.log(person._name); // Accessing private attribute (not
recommended)
console.log(person.getName()); // Accessing private attribute through a
getter
```

☞ Closures

You can use closures to create private variables within a function scope. By defining attributes within a function and returning only the necessary public methods, you can achieve encapsulation.

```
function createPerson(name, age) {
  let _name = name; // Private attribute

  return {
    getAge: function() {
      return age;
    },
    getName: function() {
      return _name;
    }
  };
}

const person = createPerson("John", 30);
console.log(person._name); // Private attribute inaccessible
console.log(person.getName()); // Accessing private attribute through a
public method
```

☞ ES6 WeakMap

You can use closures to create private variables within a function scope. By defining attributes within a function and returning only the necessary public methods, you can achieve encapsulation.

```
const privateData = new WeakMap();

class Person {
  constructor(name, age) {
    privateData.set(this, { name }); // Private attribute
    this.age = age; // Public attribute
  }

  getName() {
    return privateData.get(this).name;
  }
}

const person = new Person("John", 30);
console.log(person.name); // Private attribute inaccessible
console.log(person.getName()); // Accessing private attribute through a
method
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

Getter Setters in Object

☞ Getter Setters in Constructor Objects

```
function Person(name) {
    // Private variables
    let _name = name;

    // Getter method
    Object.defineProperty(this, 'name', {
        get: function() {
            return _name;
        },
        enumerable: true, // Make the property enumerable
        configurable: true // Make the property configurable
    });

    // Setter method
    Object.defineProperty(this, 'name', {
        set: function(value) {
            _name = value;
        },
        enumerable: true,
        configurable: true
    });
}

const person = new Person('John');
console.log(person.name); // Output: John

person.name= 'Jane';
console.log(person.name); // Output: Jane
```

☞ Getter Setters in Literal Objects

```
var person = {
    _name : "John",
    get name() {
        return this._name;
    },
    set name(name) {
        this._name = name;
    }
}

console.log(person.name); // Output: John
person.name= 'Jane';
console.log(person.name); // Output: Jane
```

☞ Getter Setters in Class

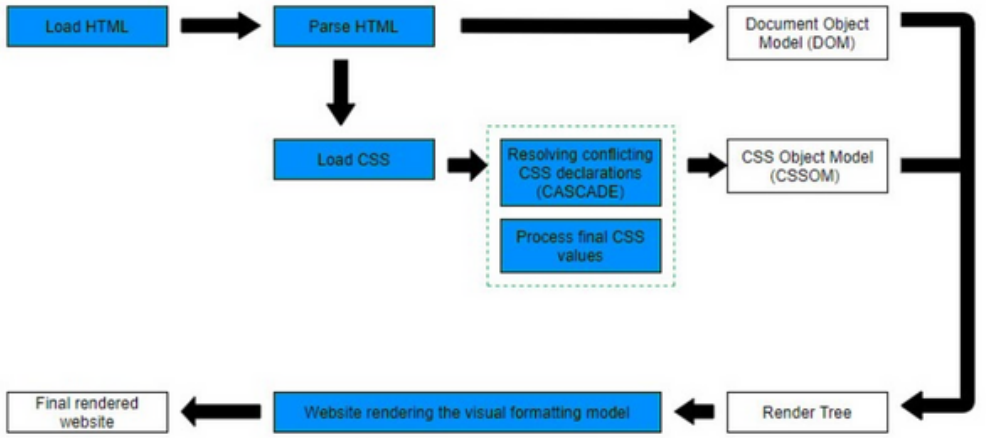
```
class Person {
    constructor(name) {
        this._name = name;
    }
    get name() {
        return this._name;
    }

    set name(name) {
        this._name = name;
    }
}

var person = new Person('Jhon');
conole.log(person.name); // Jhon
person.name = 'Jane';
conole.log(person.name); // Jane
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

DOM Manipulation



DOM stands for Document Object Model. It is a programming interface for web documents, representing the structure of HTML, XML, and XHTML documents as a tree-like model. The DOM provides a way for programs to interact with the web page dynamically and manipulate its content, structure, and styling.

In simpler terms, the DOM is a representation of a web page that JavaScript can access and modify. It treats an HTML or XML document as a structured tree of objects, where each element, attribute, and text node is represented as an object with properties and methods.

Element Selecting Methods

JavaScript provides several methods for selecting elements from the DOM (Document Object Model). These methods allow you to locate and interact with specific elements on a web page.

☞ **getElementById() – search element by element_id**

```
const element = document.getElementById("myElement");
```

☞ **getElementsByTagName() – search element by tag name (e.g., span, div)**

```
const elements = document.getElementsByTagName("div");
```

☞ **getElementsByClassName() – search element by class name**

```
const elements = document.getElementsByClassName("myClass");
```

☞ **getElementsByName() – search element by name attribute**

```
const elements = document.getElementsByName("username");
```

☞ **querySelector() – returns the first element that matches the specified selector**

```
const element = document.querySelector("#myElement .myClass");
```

☞ **querySelectorAll() – returns elements that match the specified selector**

```
const elements = document.querySelectorAll(".myClass");
```

Kavindu Samarasinghe
Software Engineer
B.Sc (First Class Honours) in Computer Science

Storages in JS

☞ Local Storage

The localStorage object allows you to store key-value pairs persistently in the browser. ***The data stored in localStorage remains available even after the browser is closed and reopened.*** You can use the setItem(key, value) method to store a value and getItem(key) method to retrieve a value.

```
// Adding data to local storage
localStorage.setItem('username', 'John');
localStorage.setItem('email', 'john@example.com');

// Retrieving data from local storage
var username = localStorage.getItem('username');
var email = localStorage.getItem('email');

console.log(username); // Output: John
console.log(email); // Output: john@example.com

// Removing data from local storage
localStorage.removeItem('email');

// Retrieving data after removal
email = localStorage.getItem('email');
console.log(email); // Output: null
```

☞ Session Storage

The sessionStorage object is similar to localStorage, but ***the data stored in sessionStorage is available only for the duration of the browser session.*** Once the browser is closed, the data is cleared.

```
// Adding data to session storage
sessionStorage.setItem('username', 'John');
sessionStorage.setItem('email', 'john@example.com');

// Retrieving data from session storage
var username = sessionStorage.getItem('username');
var email = sessionStorage.getItem('email');

console.log(username); // Output: John
console.log(email); // Output: john@example.com

// Removing data from session storage
sessionStorage.removeItem('email');

// Retrieving data after removal
email = sessionStorage.getItem('email');
console.log(email); // Output: null
```

☞ Cookies

Cookies are small text files that can be used to store data in the browser. ***They have an expiry date and time,*** and can be read by both the client-side JavaScript and the server-side code. You can use the document.cookie property to set and retrieve cookies.

```
// Function to set a cookie
function setCookie(name, value, days) {
  var expires = '';
  if (days) {
    var date = new Date();
    date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
    expires = '; expires=' + date.toUTCString();
  }
  document.cookie = name + '=' + value + expires + '; path=/';
}

// Function to get the value of a cookie
function getCookie(name) {
  var cookieName = name + '=';
  var cookies = document.cookie.split(';');

  for (var i = 0; i < cookies.length; i++) {
    var cookie = cookies[i];
    while (cookie.charAt(0) === ' ') {
      cookie = cookie.substring(1);
    }
    if (cookie.indexOf(cookieName) === 0) {
      return cookie.substring(cookieName.length, cookie.length);
    }
  }
  return null;
}

// Function to remove a cookie
function removeCookie(name) {
  document.cookie = name + '=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/;';
}

// Setting a cookie
setCookie('username', 'John', 7);

// Getting the value of a cookie
var username = getCookie('username');
console.log(username); // Output: John

// Removing a cookie
removeCookie('username');

// Getting the value after removal
username = getCookie('username');
console.log(username); // Output: null
```

Cookies with jQuery

```
// Setting a cookie
$.cookie('username', 'John', { expires: 7, path: '/' });

// Getting the value of a cookie
var username = $.cookie('username');
console.log(username); // Output: John

// Removing a cookie
$.removeCookie('username');

// Getting the value after removal
username = $.cookie('username');
console.log(username); // Output: null
```

☞ IndexedDB

IndexedDB is a more advanced storage mechanism that provides a transactional database system for storing structured data in the browser. It allows you to store large amounts of data, perform complex queries, and supports indexes for efficient data retrieval. U***sing IndexedDB requires more code compared to local storage or session storage***, ***so it's more suitable for complex data scenarios.***

Internet Technologies

JavaScript Inheritance

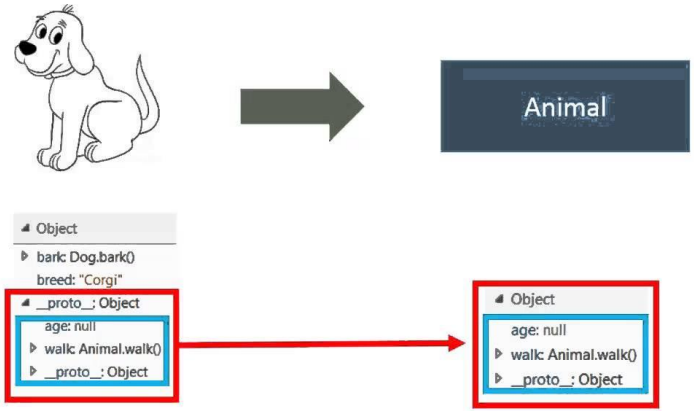
Kavindu Samarasinghe

B.Sc (Hons) in Computer Science (1st class), Graduate Dip in Software Engineering, Certificate in Digital Marketing (APIDM)

JavaScript Inheritance

JS inheritance is the method through which the objects inherit the properties and the methods from the other objects.

It enables code reuse and structuring of relationships between objects, creating a hierarchy where a child object can access features of its parent object.



Inheritance in JavaScript can be achieved in the following ways:

- Prototypal Inheritance
- Classical Inheritance
- Functional Inheritance

Prototypal Inheritance

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.sound = function() {
    console.log("Some generic sound");
};

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.sound = function() {
    console.log("Woof! Woof!");
};

const myDog = new Dog("Buddy", "Labrador");
myDog.sound();  // Outputs: Woof! Woof!
```

Classical Inheritance

Introduced in ES6 with the class keyword. Uses a class-based approach similar to other programming languages like Java.

JavaScript ES6 classes support the **extended keyword** to perform class inheritance.

```
class automobile {
    constructor(name, cc) {
        this.name = name;
        this.cc = cc;
    }
    engine() {
        console.log(`${this.name}
    has ${this.cc} engine`);
    }
}

class car extends automobile {
    engine() {
        console.log(this.name,
            "has ", this.cc, "cc engine");
    }
}

let carz = new car('Rex', "1149");
carz.engine();
```

Inheritance using the super keyword - Class Inheritance

```
// Inheritance using super keyword in JS
class Automobile {
  constructor(name) {
    this.name = name;
  }

  engine() {
    console.log(this.name,
      "has ", this.cc, "cc engine");
  }
}

class Car extends Automobile {
  constructor(name, cc) {
    super(name);

    // Additional properties for
    // the Car class
    this.cc = cc;
  }

  engine() {
    // the 'engine' method of the parent
    // class using 'super'
    super.engine();

    console.log(this.name,
      "has ", this.cc, "cc engine");
  }
}

let carz = new Car('Rexton', '1500');
carz.engine();
```

Functional Inheritance

Objects inherit properties and methods from other objects through function constructors. It uses functions to create objects and establish relationships between them.

```
function Animal(name) {
  const obj = {};
  obj.name = name;

  obj.sound = function() {
    console.log("Some generic sound");
  };

  return obj;
}

function Dog(name, breed) {
  const obj = Animal(name);
  obj.breed = breed;

  obj.sound = function() {
    console.log("Woof! Woof!");
  };

  return obj;
}

const myDog = Dog("Buddy", "Labrador");
myDog.sound(); // Outputs: Woof! Woof!
```

Multi-level Inheritance

Self References:

https://forum.freecodecamp.org/t/multi-level-inheritance-in-javascript/245532/3