JavaScript

- JavaScript is a **programming language** used to create dynamic content for websites
- It is a **lightweight**, **cross-platform,** and **single-threaded** programming language.
- JavaScript is an **interpreted** language that executes code line by line providing more flexibility.
- **Just-In-Time (JIT)** compilation in JavaScript is a dynamic optimization technique that improves performance by compiling frequently executed code into machine code during runtime.

How to use the Javascript?

1. Inline
2. Internal
3. External

```
<button onclick="alert('Button Clicked!')"> Click Here
</button>
```

```
<head>
 <script>
 Alert("welcome js")
</script>
</head>
```

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

JavaScript can "display" data in different ways:

- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().
- window.print()

JavaScript Comments

Single line comments start with //.

Multi-line comments start with /* and end with */.

Inputs

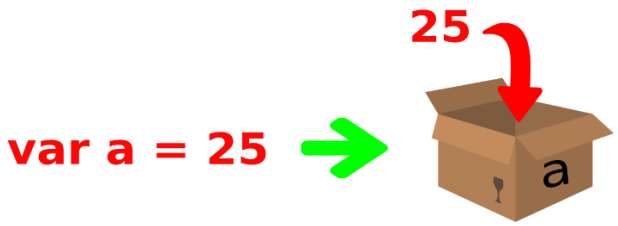
```
// prompt("Enter your Name?")

confirm("Do you want to Logout?")
```

Variables

Variables are Containers for Storing Data

Variables can be declared in 3 ways:

- Using var
- Using let
- Using const



JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

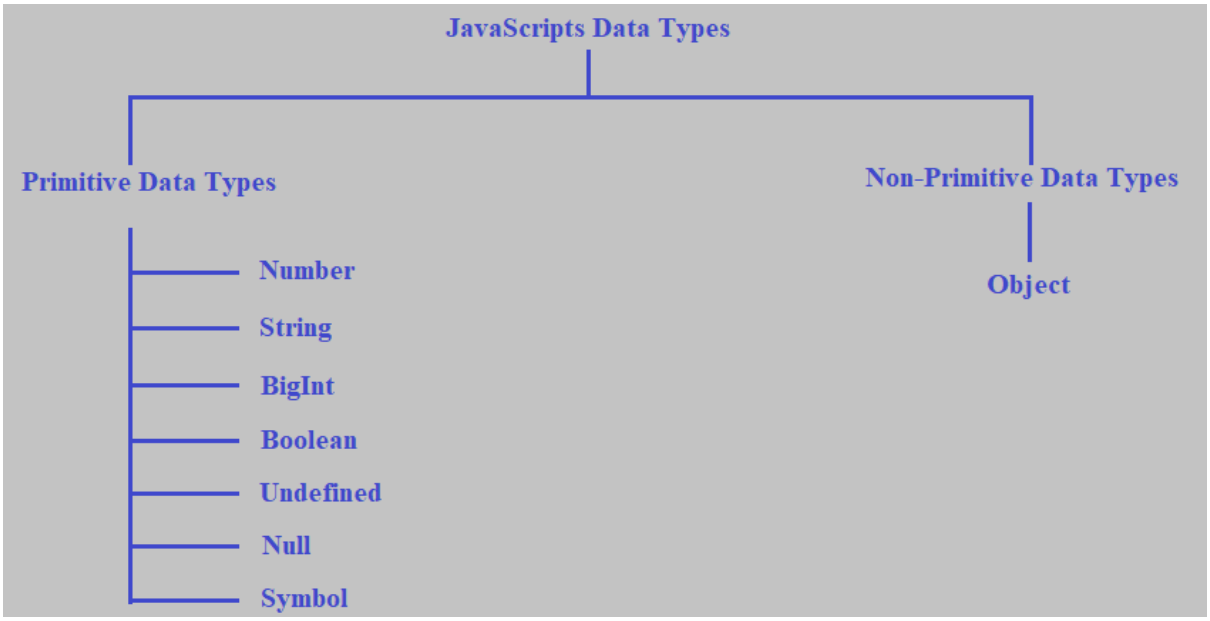
These unique names are called **identifiers**.

The general rules for constructing names for variables

- Names can contain **letters, digits and underscores (_) and dollar ($)** signs.
- Names must begin with **a letter**.

- Names are **case sensitive** (**y and Y are different variables**).
- Reserved words (like JavaScript keywords) cannot be used as names.

JavaScript Data Types



JavaScript's conditional statements

- **JavaScript if-statement**
- **JavaScript if-else statement**
- **JavaScript nested-if statement**
- **JavaScript if-else-if ladder statement**

JavaScript if-statement

It is a conditional statement used to decide whether a certain statement or block of statements will be executed or not.

Syntax:

```
if(condition) // true
{
  // Statements to execute if
  // condition is true
}
```

JavaScript if-else statement

the if...else statement is a conditional statement that executes a block of code based on whether a specified condition is true or false.

```
if (condition) {
  // Code to execute if the condition is true
} else {
  // Code to execute if the condition is false
}
```

nested-if statement

JavaScript allows us to nest if statements within if statements

we can place an if statement inside another if statement

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

if-else-if ladder statement

Use the else if statement to specify a new condition if the first condition is false.

Multiple condition

Syntax

```
if (condition1) {
  //  block of code to be executed if condition1 is true
```

```
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is false
}
```

switch Statement

The **JavaScript switch statement** evaluates an expression and executes a block of code based on matching cases

Syntax

```
switch (expression) {
    case value1:
        // code block 1;
        break;
    case value2:
         // code block 2;
        break;
   ...
    default:
      // default code block;
}
```

JavaScript Loops

Loops in JavaScript are used to **reduce repetitive tasks by repeatedly executing a block of code as long as a specified condition is true. This makes code more concise and efficient.**

1. While Loop
2. Do .. while Loop
3. For

4. For ..in
5. For .. of
6. Foreach()

While:

It executes as long as the condition is true. It can be thought of as a repeating if statement.

Syntax

```
// initialization
while (condition) {
    // Code to execute
    // increment / decrement
}
```

Do .. while

It is similar to while loop except it executes the code block at least once before checking the condition.

Syntax

```
 //initialization
do {
    // Code to execute
} while (condition);
```

JavaScript for Loop

It is repeats a block of code a **specific number of times**. It contains initialization, condition, and increment/decrement in one line.

Syntax

```
for (initialization; condition; increment/decrement) {
    // Code to execute
}
```

Break	
Keyword ‘break’ is used	Keyword ‘c
It is used to terminate the loop	It is used t iteration ir
It is used in switch case and looping statements	It is used i only.
In break statement, control transfers outside the loop	In continue remains in

Functions in JavaScript

Functions in JavaScript are reusable blocks of code designed to perform specific tasks.

They allow you to organize, reuse, and modularize code

Why Functions?

- Functions can be used multiple times, reducing redundancy.
- Break down complex problems into manageable pieces.
- Manage complexity by hiding implementation details.
- Can call themselves to solve problems recursively.

Types of Functions:

==========

1. Function Declaration
2. Function Expression
3. Arrow Function
4. Anonymous Functions
5. Callback Functions
6. Immediately Invoked Function Expressions (IIFE)
7. Async Function
8. Recursion function

Objects

An object in JavaScript is a **data structure used to store related data collections**.

It stores data as key-value pairs,

where **each key is a unique identifier** for the associated value.

Objects are dynamic, which means the properties can be added, modified, or deleted at runtime.

There are two primary ways to create an object in JavaScript: **Object Literal** and **Object Constructor**.

1. Creation Using Object Literal
initialize an object with curly braces **{}**, setting properties **as key-value pairs.**

Syntax:

```
Let variable= {
        Key1: value1,
      Key2: value2,
      Key3: value3
}
```

```
let obj = {
```

```
    name: "Sourav",
    age: 23,
    job: "Developer",
  studentDetails: function () {
      return this.name + " " + this.class
        + " " + this.section + " ";
    }
  };
  console.log(obj);
```

2. Creation Using new Object() Constructor

```
  let obj = new Object();
  obj.name= "Sourav",
  obj.age= 23,
  obj.job= "Developer"

  console.log(obj);
```

Basic Operations

1. Accessing Object Properties

dot notation or bracket notation

2. Modifying Object Properties

```
  obj.age = 23;
```

3. Adding Properties to an Object

```
  obj.color = "Red";
```

4. Removing Properties from an Object

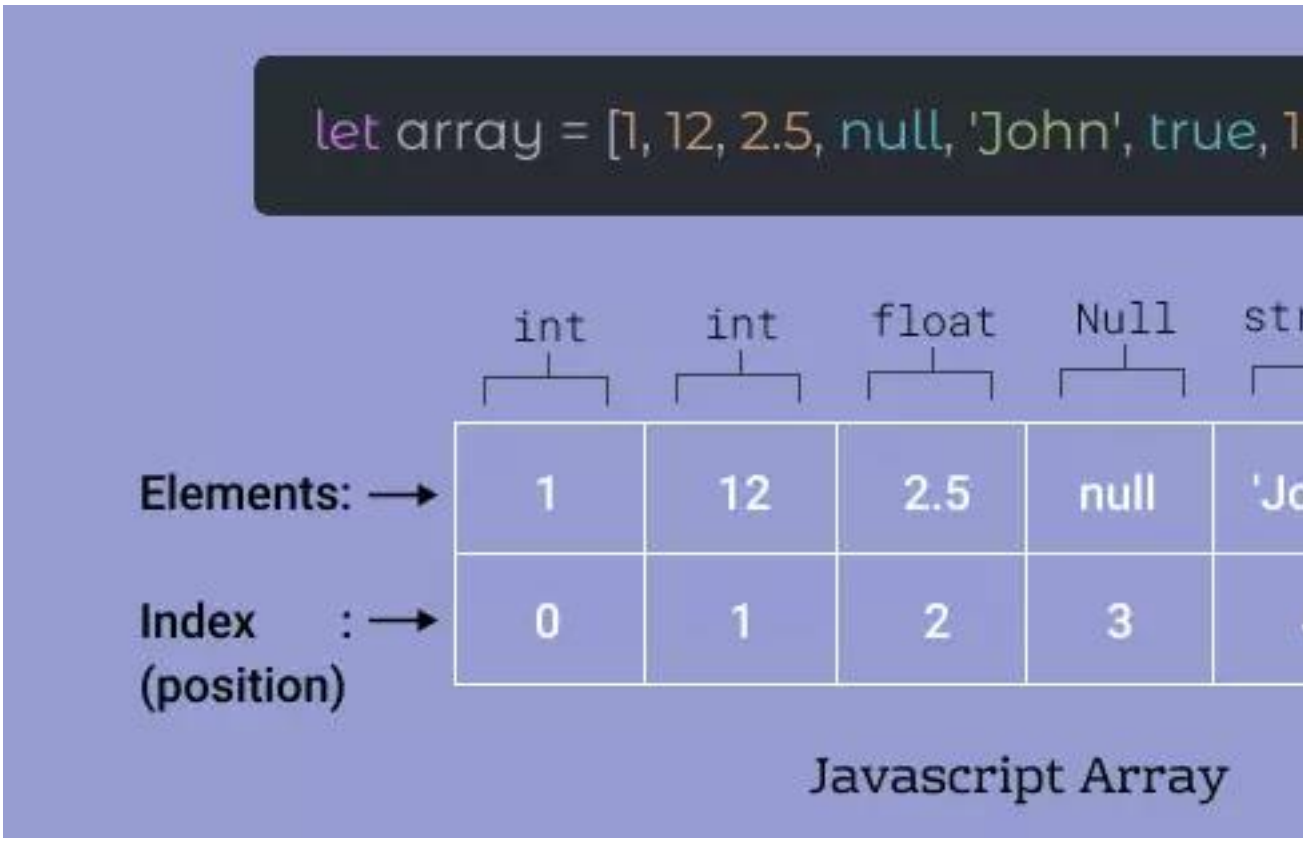
delete operator

delete obj.color;

An array is an ordered list of values.

Each value is called an element, and each element has a numeric position in the array, known as its index

the first element is at index 0,



1. Create Array using Literal

Creating an array using array literal involves using square brackets [] to define and initialize the array.

```
let a = [];

console.log(a);

// Creating an Array and Initializing with Values

let b = [10, 20, 30];
```

```
console.log(b);
```

2. Create using new Keyword (Constructor)

The "**Array Constructor**" refers to a method of creating arrays by invoking the Array constructor function.

```
let a = new Array(10, 20, 30);

console.log(a);
```

---

1. Accessing Elements of an Array

Any element in the array can be accessed using the index number. The index in the arrays starts with 0.

```
let a = ["HTML", "CSS", "JS"];


// Accessing Array Elements

console.log(a[0]);

console.log(a[1]);
```

Access first element & last element a. length -1

Modifying the Array Elements

```
let a = ["HTML", "CSS", "JS"];

console.log(a);

a[1]= "Bootstrap";

console.log(a);
```

Delete array elements

Delete a[0]

Array Destructuring / UnPack

allows for the unpacking of values from arrays, or properties from objects, into distinct variables

```
const numbers = [10, 20, 30];
const [a, b, c] = numbers; // a = 10, b = 20, c = 30

// Skipping elements
const [first, , third] = numbers; // first = 10, third = 30
```

Const names = [‘siva’,’raja’,’mani’]

Const [a,...rest] = name;

Spread operator ? ...

Array functions

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

1. fruits.length; The length property **returns the length** (size) of an array

   fruits.length = 2; ex : Banana,Orange

2. toString() method **returns the elements of an array as a comma separated string**. Ex: fruits.toString();
3. fruits.at(2); fruits[2] element of fruits using at():
4. The join() method also **joins all array elements into a string. separator**: fruits.join(" * ");
5. The pop() **method removes the last element from an array**: fruits.pop();
6. The push() method **adds a new element to an array (at the end):** fruits.push("Kiwi");
7. The shift() method **removes the first array element and "shifts" all other elements to a lower index**. fruits.shift();

8. The unshift() method adds a **new element to an array (at the beginning), and "unshifts" older elements**: fruits.unshift("Lemon");
9. Using delete() leaves undefined holes in the array.
10. delete fruits[0]; , show undefined
11. The concat() method creates a new array by merging (concatenating) existing arrays:
12. Array1.concat(array2);
13. **console.log(names.concat([1,2,3]))**

for

For ...in => get the array index

For ...of => get the array elements

```
let len =  names.length
 //  console.log(len)
 // iteration
 for(let i=3;i<len;i++){
    document.writeln(names[i],"<br>")
 }
 /*
  for(let a  in array){

  }
 */
 for(let i  in names){
    document.writeln(names[i])
 }
 /*
 for(let e of array){
 }
 */
 for(let ele of names){
    console.log(ele)
 }
```

```
let num = [10,20,30,40,50];
```

1. **sum of array +=**
2. **count the element**

```
let studentName = "Denish@12345";

 console.log(studentName[0])
 console.log(studentName[1])
 console.log(studentName[2])
 console.log(studentName[3])
 console.log(studentName[4])

//  a-z count the letters  6
//  0-9 count the numbers  5
//   count special charactors in 1
//  a,e,i,o,u
//  count the vowels 2
```

1.The splice() method adds new items and also remove and update to an array.

2.The slice() method slices out a piece of an array.

Must add the two parameters:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.splice(0,1); //

fruites.splice(1,2);//
```

fruites.splice(1,0,”Sweet”,”jaggery”) //
Banana,Sweet,Jaggery,Orange,Apple, Mango

fruites.splice(1,2,”Sweet”,”jaggery”) //
Banana,Sweet,Jaggery,Mango

==========
==========
==========

The method then selects elements from the start argument, and up to (but not including) the end argument.

const citrus = fruits.slice(1); /
/Orange,Lemon,Apple,Mango

const citrus = fruits.slice(1,3); // Orange,Lemon

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
```

```
numbers = [3,5,7,8,10,20,60,23]
```

```
const numbers.find((value,index,array)=>{return
value>18})
```

Array indexOf()	Returns the **first position** of an element value	
Array lastIndexOf()	Returns the **last position** of an element value	Ex
Array includes()	Returns **true** if an element value is present in an array	E

Array find()	Returns the value of the **first element** that passes a test	E el
Array findIndex()	Returns **the index of the first element** that passes a test	Ex in
Array findLast()	Returns the value of the **last element that passes** a test	co le
Array findLastIndex()	Returns the **index of the last element** that passes a test	co le

Array Sort Methods

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

sort() method sorts an array alphabetically: only string

```
fruits.sort();
```

The reverse() method reverses the elements in an array:

```
fruits.reverse();
```

toSort(); toReversed();

Number sort

```
points.sort(function(a, b){return a - b});
```

```
array.forEach(()=>{


})
```

forEach()
Executes a callback on each array element and returns undefined. It’s ideal for **side effects** (like logging, updating variables, DOM actions). Do *not* use it if you want to produce a new array

filter()

Returns a new array with only those items that satisfy a given condition. Keeps or removes each element based on true/false result

map()

each element and returns a **new array** of the same length. Use it when you want to transform values. The original array is not mutated.

reduce()

Processes all elements to produce a **single output value**, using an accumulator that carries across iterations. **Useful for summarizing arrays into sums, counts, or aggregated objects.**

Set() =>unique records

JavaScript Strings

A JavaScript String is a **sequence of characters**.

Let a = “abc”

Types:

String Literals

String Constructor

Template Literals (String Interpolation)

Template literals allow you to embed expressions within backticks (`) for dynamic string creation, making it more readable and versatile.(Interpolation)

Template String provide an easy way to interpolate variables and expressions into strings. ${...}

```
let s1 = 'abcd'; // recommended
console.log(s1);
let s = new String('abcd');
console.log(s);
let s2 = `You are learning from ${s1}`;

console.log(s2);
```

1. Finding the length of a String

```
let s = 'JavaScript';
let len = s.length;
```

2. String Concatenation

```
let s1 = 'Java';
let s2 = 'Script';
```

```
let res = s1 + s2;

 let txt = s1.concat(s2)

console.log(txt)
```

3. Escape Characters

```
\' - Inserts a single quote
\" - Inserts a double quote
\\ - Inserts a backslash
const s1 = "\'SDLC\' is a learning portal";
const s2 = "\"SDLC\" is a learning portal";

const s3 = "\\SDLC\\ is a learning portal";

console.log(s1);

console.log(s2);

console.log(s3);
```

4. Find Substring of a String

```
let s1 = 'JavaScript Tutorial';

let s2 = s1.substring(0, 10);
```

5. Convert String to Uppercase and Lowercase

```
let uCase = s.toUpperCase();

let lCase = s.toLowerCase();
```

==========

The charAt() method returns the character at a specified index (position) in a string: text.charAt(0); name.at(2); text[0];

The charCodeAt() method returns the ASCII code of the character at a specified index in a string:
text.charCodeAt(0)

==========
==========

There are 3 methods for extracting a part of a string:

- slice(*start*, *end*)
- substring(*start*, *end*)
- substr(*start*, *length*)

- slice() extracts a part of a string and returns the extracted part in a new string.

- The method takes 2 parameters: start position, and end position (end not included).

```
let text = "Apple, Banana, Kiwi";
let part = text.slice(7, 13);
```

```
let part = text.slice(7);
```

```
let part = text.slice(-12);
```

```
let part = text.slice(-12, -6);
```

substring() is similar to slice().

The difference is that start and end values less than 0 are treated as 0 in substring().

```
let part = str.substring(7, 13);
```

==========
==========
==========

concat() joins two or more strings:

The trim() method removes whitespace from both sides of a string: text1.trim(); trimStart(), trimEnd()

The padStart() method pads a string from the start.
let text = "5"; let padded = text.padStart(4,"0"); Ex : 0004

The padEnd() method pads a string from the end.

The repeat() method returns a string with a number of copies of a string.

The repeat() method returns a new string. text.repeat(2);

==========
==========
==========
==

The replace() method replaces a specified value with another value in a string:

```
 let text = "Please visit Microsoft!";
let newText = text.replace("Microsoft", "AROSPACE");

let newText = text.replace(/MICROSOFT/i, "DEX"); case insensitive

let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace(/Microsoft/g, "HEVEN"); global match

text = text.replaceAll("cats","dogs");
```

A string can be converted to an array with the split() method: text.split("
") text.split("|") text.split(",")

The try...catch...finally statement in JavaScript is used for error handling. It allows the execution of code that might throw an error, and provides a mechanism to handle that error gracefully, as well as execute code regardless of whether an error occurred or not.

- **try**:

  This block contains the code that is being attempted. If an error occurs within this block, the control flow immediately jumps to the catch block.

- **catch**:

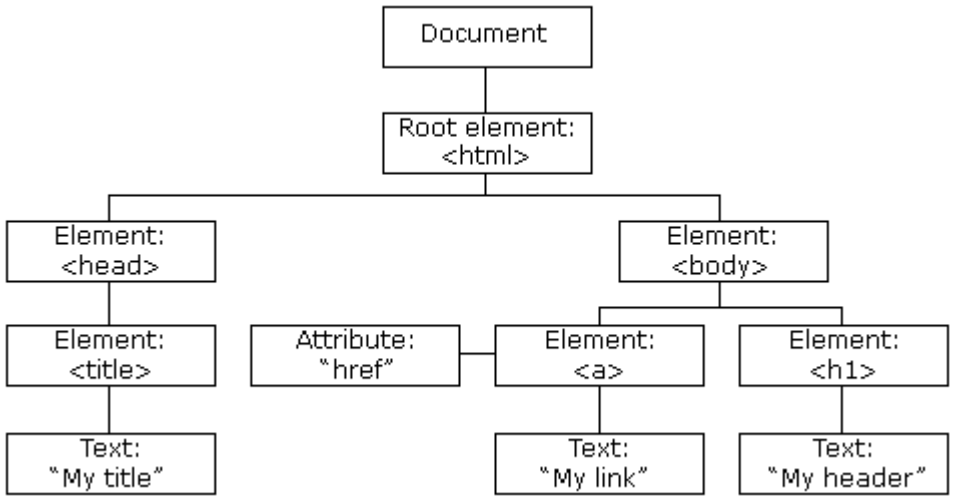
  This block contains the code that is executed if an error is thrown in the try block. It receives an error object as an argument, which can be used to get information about the error.

- **finally**:

  This block contains code that is always executed after the try and catch blocks, regardless of whether an error was thrown or caught. It is often used for cleanup operations, such as closing files or releasing resources.

```
try {
  // Code that might throw an error
  let result = someFunctionThatMightFail();
  console.log("Result:", result);
} catch (error) {
  // Handle the error
  console.error("An error occurred:", error.message);
} finally {
  // Code that always executes
  console.log("Finally block executed");
}
```

JavaScript HTML DOM(Document Object Model)



JavaScript can access and change all the elements of an HTML document.

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

DOM Methods

Method	**Description**
document.getElementById(*id*)	Find an element
document.getElementsByTagName(*name*)	Find elements by

document.getElementsByClassName(*name*)	Find elements by

Property	**Description**
element.innerHTML = *new html content*	Change the inne
element.*attribute* = *new value*	Change the attri
element.style.*property* = *new style*	Change the style
Method	**Description**
element.setAttribute*(attribute, value)*	Change the attri

JavaScript supports a wide range of events:

- **Mouse Events**: click, dblclick, mousemove, mouseover,mouseenter,
- mouseout,
- **Keyboard Events**: keydown, keypress, keyup, input.
- **Form Events**: submit, change, focus, blur.
- **Window Events**: load, resize, scroll, unload.
- **Clipboard Events**: copy, cut, paste.
- **Touch Events**: touchstart, touchmove, touchend.

The Browser Object Model (BOM)

- window.innerHeight - the inner height of the browser window (in pixels)

- window.innerWidth - the inner width of the browser window (in pixels)

- window.open() - open a new window
- window.close() - close the current window
- window.moveTo() - move the current window
- window.resizeTo() - resize the current window

```
<h2>Window Manipulation Example</h2>

  <button onclick="openWindow()">Open New
Window</button>

  <button onclick="moveWindow()">Move
Window</button>

  <button onclick="resizeWindow()">Resize
Window</button>

  <button onclick="closeWindow()">Close
Window</button>



  <script>

    let newWindow;



    function openWindow() {

      newWindow = window.open("", "myWindow",
"width=400,height=300");

      newWindow.document.write("<p>This is a new
window!</p>");

    }
```

```
function moveWindow() {
  if (newWindow) {
    newWindow.moveTo(200, 150);
  } else {
    alert("Please open a window first.");
  }
}

function resizeWindow() {
  if (newWindow) {
    newWindow.resizeTo(600, 400);
  } else {
    alert("Please open a window first.");
  }
}

function closeWindow() {
  if (newWindow) {
    newWindow.close();
  } else {
    alert("Please open a window first.");
```

```
    }
  }
</script>
```

- screen.width
- screen.height
- screen.availWidth
- screen.availHeight
- screen.colorDepth
- screen.pixelDepth

```
function displayScreenInfo() {
    const screenInfo = `
      <strong>Screen Width:</strong> ${screen.width} px<br>
      <strong>Screen Height:</strong> ${screen.height} px<br>
      <strong>Available Width:</strong> ${screen.availWidth} px<br>
      <strong>Available Height:</strong> ${screen.availHeight} px<br>
      <strong>Color Depth:</strong> ${screen.colorDepth} bits<br>
      <strong>Pixel Depth:</strong> ${screen.pixelDepth} bits
    `;
    document.getElementById('screenInfo').innerHTML = screenInfo;
```

}

Window.location

==========

The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.

- window.location.href returns the href (URL) of the current page
- window.location.hostname returns the domain name of the web host
- window.location.pathname returns the path and filename of the current page
- window.location.protocol returns the web protocol used (http: or https:)
- window.location.port
- window.location.assign() loads a new document
- window.location.assign("https://www.w3schools.com")

Window History

The window.history object can be written without the window prefix.

To protect the privacy of the users, there are limitations to how JavaScript can access this object.

Some methods:

- history.back() - same as clicking back in the browser
- history.forward() - same as clicking forward in the browser

window.confirm("*sometext*");

- setTimeout(*function, milliseconds*)
  Executes a function, after waiting a specified number of milliseconds.

- setInterval(*function, milliseconds*)
  Same as setTimeout(), but repeats the execution of the function continuously.

localStorage , sessionStorage

```
<input id="id1" type="number" min="100" max="300" re
quired>
<button onclick="myFunction()">OK</button>

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

<script>
function myFunction() {
  const inpObj = document.getElementById("id1");
  if (!inpObj.checkValidity()) {
    document.getElementById("demo").innerHTML =
inpObj.validationMessage;
  }
}
</script>
```

Classes and Objects in JavaScript

Class is nothing but a **blueprint** for an object of it.

The Constructor Method

The constructor method is a special method in JavaScript with the exact name ‘constructor.’

It is automatically **executed when a new object is created from a class.**

Its primary **purpose is to initialize object properties,** allowing **you to define the initial state of an object during its instantiation.**

JavaScript Class Methods Syntax:

```
class Name {
   constructor(var) {
          this.var = var;

   }
    // defining method
    method() {
       //Code Here
    }
}

class Name {
    constructor(var) {
       this.var = var;
    }
    // defining getter method
    get method() {
       //Code Here
    }
    // defining setter method
    set method(value) {
       this.var = value;
    }
}
```

Class Inheritance

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}

let myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML =
myCar.show();
```

By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

Static class methods are defined on the class itself.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}

const myCar = new Car("Ford");
```

A Fetch API Example

The example below fetches a file and displays the content:

How Fetch API Works?

- A request is sent to the specified URL.
- The server processes the request and sends a response.
- The response is converted to JSON (or another format) using .json().
- Errors are handled using .catch() or try-catch blocks.

Common HTTP Request Methods in Fetch API

- **GET:** This request helps to retrieve some data from another server.
- **POST**: This request is used to add some data onto the server.
- **PUT:** This request is used to update some data on the server.
- **DELETE:** This request is used to delete some data on the server.

```
fetch(url, options)
  .then(response => {
    // Handle the response
  })
  .catch(error => {
    // Handle errors
  });

fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

```
fetch('https://api.example.com/data')
    .then(response => {
        if (response.ok) {
            return response.json();
        } else {
            throw new Error('Network response was not ok');
        }
    })
    .then(data => console.log(data))
    .catch(error => console.error('There was a problem with the fetch operation:', error));
```

Handling Responses

- response.ok: A boolean indicating if the response was successful (status in the range 200-299).
- response.status: The HTTP status code of the response.
- response.json(): Parses the response body as JSON.

- response.text(): Parses the response body as text.

```
async function getP() {
    try {
        const response = await
fetch('https://fakestoreapi.com/products');
        if (response.ok) {
            const data = await response.json();
            console.log(data);
        } else {
            throw new Error('Failed to fetch data');
        }
    } catch (error) {
        console.error('Error:', error);
    }
}
getP()
```

- **async function getP()**: This defines an asynchronous function, meaning it can handle tasks like fetching data without blocking the rest of the program.
- **await fetch()**: The await keyword pauses the function until the fetch() request is complete, so the data can be used right after it's retrieved.
- **response.ok**: Checks if the fetch request was successful by ensuring the response status is in the 200-299 range.
- **await response.json()**: If the response is successful, it converts the data from the server (usually in JSON format) into a JavaScript object.
- **try/catch block**: Catches any errors that may happen (like network problems) and logs them, preventing the program from crashing.

Handling Different Request Methods (POST, PUT, DELETE)