JavaScript

JavaScript

JavaScript

JavaScript is a **high-level scripting language** used to create **interactive and dynamic web pages.**

It runs on both the **client side (Browser)** meaning we can directly execute the js file on the browser, and **server side** (nodejs)

It is the **native programming language of the browser. Because there is js engine in every browser**

Characteristics of JS

- **Scripting language** : It is used to automate tasks , first js is loaded in the browser for a webpage
- after that for every action of the user javascript will be executed in the browser side(client side) and it will affect the UI accordingly.

- **Dynamically typed language** : JavaScript allows variables to hold any data type.

```
let a = 10  // initially a number
a = "JavaScript"  // now a string f"
```

- **High level language** : which is easily understandable by us human beings
- **Weakly / Loosely typed language :** It doesn't perform strict type checking.

```
console.log( "5" + 5 )   // Outputs  "55"  ( string concatenation )
console.log( "5" - 2 )   // Outputs  3  ( string converted to number )
```

- **Interpreted Language / Synchronous / Single Threaded**

Interpreted → it can be executed directly to machine code , prior compilation is not needed (No separate compiled file is created).
Synchronous → It will be executing all the tasks in the order they are written .
Single Threaded → It can execute one task at a time , it cannot execute multiple tasks simultaneously.

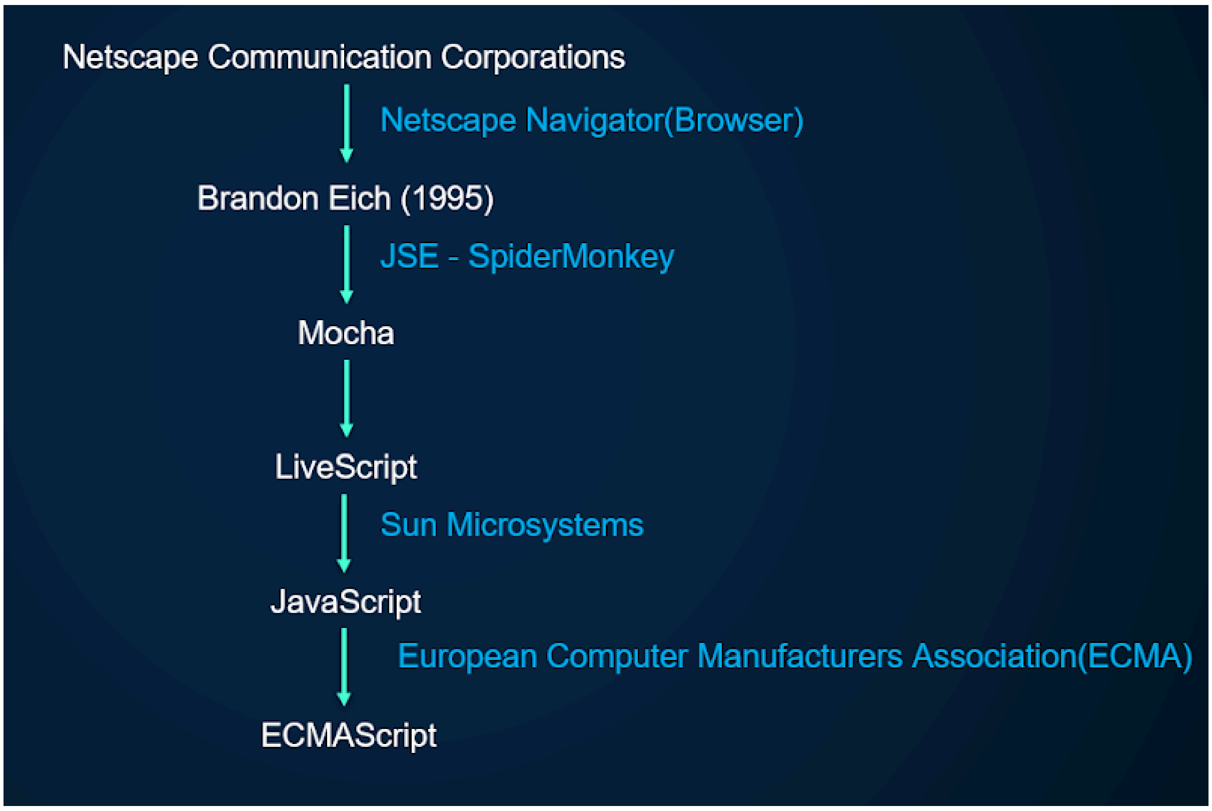
- **Object Based Language :** Everything in JavaScript is considered as an object like arrays, function etc.

- **Object Oriented Programming Language :** JavaScript was originally object-based, but now it supports **Object oriented programming** through **ES6+ features** like **classes** and **inheritence.**

- **Light-weight Language** :

JavaScript is considered lightweight due to its simple syntax, low CPU usage, and ease of implementation, especially in web browser

History Of JavaScript


Netscape Communication Corporations
Netscape Navigator(Browser)
Brandon Eich (1995)
JSE - SpiderMonkey
Mocha
LiveScript
Sun Microsystems
JavaScript
European Computer Manufacturers Association(ECMA)
ECMAScript

Most Popular Version of JavaScript

- **ES6 (2015)** → Added **let, const, arrow functions, classes**, and many other modern features.

Java v/s JavaScript

Java	JavaScript
Programming language	Programming and Scripting language
Strictly typed	Weekly / Loosely typed
Statically typed	Dynamically typed
Runs on Java Virtual Machine	Runs on all the browsers
Both compiled and interpreted language	Only interpreted language

How JavaScript Engine Works

Parsing (handled before interpreter/compiler)
Source code is converted into an **Abstract Syntax Tree (AST)** by the parser.

Interpreter (Ignition in V8)
The AST is passed to the **interpreter**, which quickly translates it into **bytecode**

Execution Starts (Using Bytecode)
The bytecode is executed by the **interpreter**. At this stage:

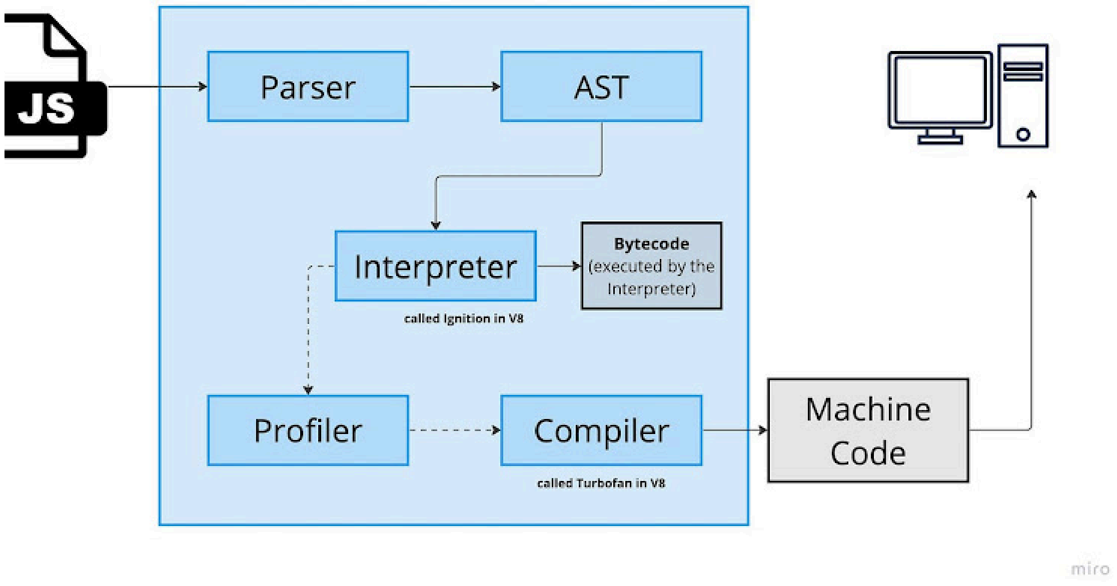
- Variables are allocated.
- Functions are invoked.
- Loops and conditionals are run.
- Memory is managed via heap and stack.

Profiler
While the interpreter runs the code, the engine **monitors** (profiles) the execution to identify **hot code** (code that's run frequently).

Optimization via JIT Compiler (Turbofan in V8)
If some functions or loops run **multiple times**, the engine flags them as "hot" and sends them to the **JIT (Just-In-Time) compiler**:

- This JIT compiler (Turbofan) generates highly optimized **machine code**.
- This optimized code **replaces** the original bytecode for faster future execution.

Execution Continues with Machine Code
Now, the engine uses the **machine code** version for that hot path, resulting in a major performance boost.

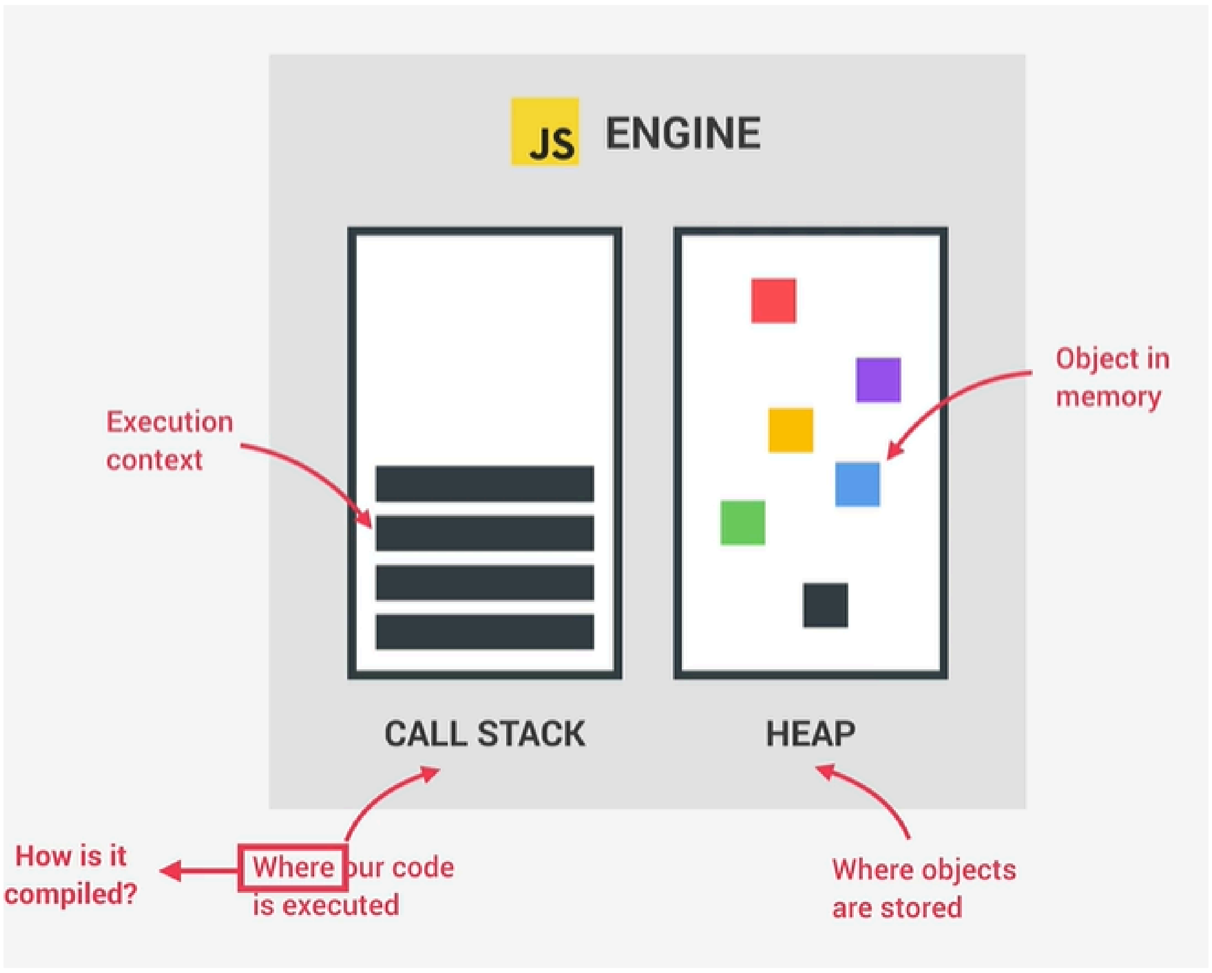


Execution

- JavaScript code is executed using **two main memory areas**:

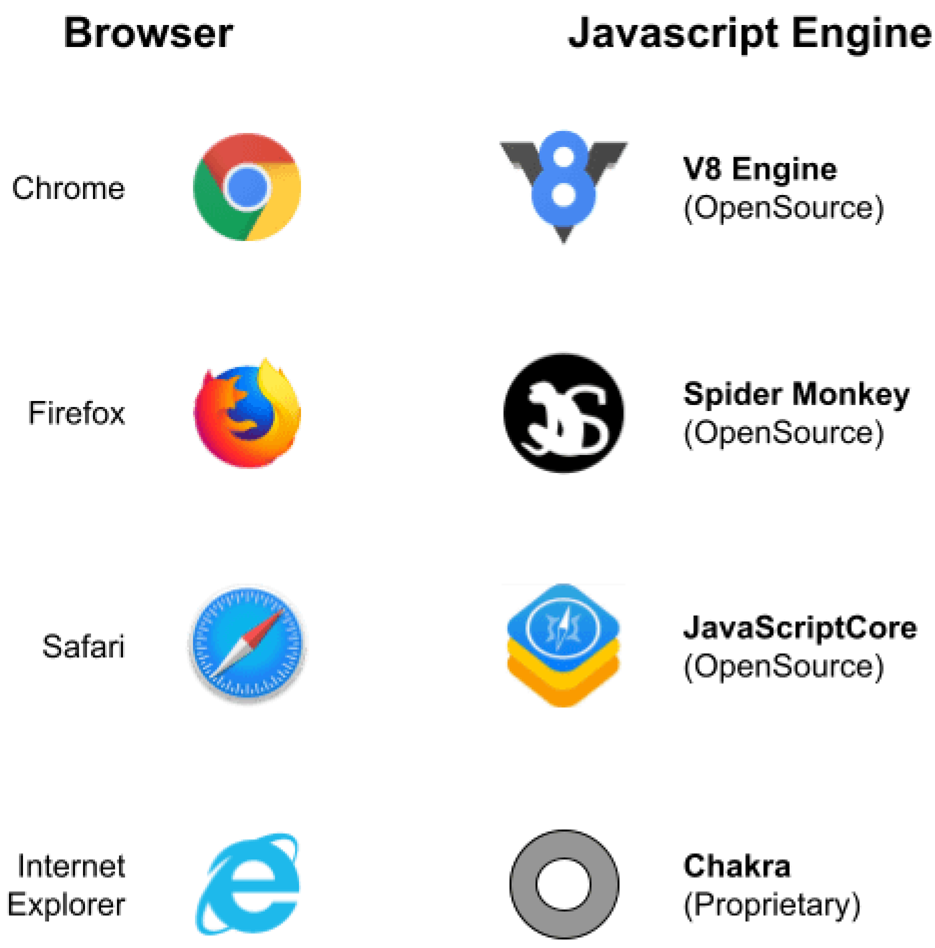
  1. **Call Stack** – Where function calls and execution happen.
  2. **Heap Memory** – Where objects are stored.

- The execution happens inside the **Call Stack**, and objects are stored in the **Heap**.



Engine of Different Browsers

- Every Browser is having a JavaScript Engine



Browser	Javascript Engine
Chrome	V8 Engine (OpenSource)
Firefox	Spider Monkey (OpenSource)
Safari	JavaScriptCore (OpenSource)
Internet Explorer	Chakra (Proprietary)

Ways to Execute JavaScript

JavaScript code can be written in two ways:

1. **Internal JavaScript**

2. **External JavaScript**

Internal JavaScript

JavaScript code is written within the same HTML file inside the <script> tag.

Example:

```
<!DOCTYPE html>
<html>
<head>
   <title>Internal JavaScript</title>
</head>
<body>
```

```
    <script>
      var sname = "Akshay";
      console.log(sname);
    </script>
  </body>
  </html>
```

External JavaScript

JavaScript code is written in a separate file, and the HTML file links to it using the <script> tag with the src attribute.

Example:

index.html

```
<!DOCTYPE html>
<html>
<head>
   <title>External JavaScript</title>
</head>
<body>
   <script src="code.js"></script>
</body>
</html>
```

code.js

```
var sname = "Akshay";
console.log(sname);
```

defer

Defer is used to stop the execution of the javascript until complete html is loaded . We must use defer if we are attaching the js file in the head section.

Example

script.js

```
console.log("Hey i am script");
```

second.js

```
console.log("Hey i am second");
```

Index.html → without defer keyword

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <script src="./script.js"></script>
  </head>
  <body>
    <script src="./second.js"></script>
  </body>
</html>
```

Output

```
Hey i am script
Hey i am second
```

Here both the file are getting executed in the same order as they have been written.

Index.html → with defer keyword

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <script src="./script.js" defer></script>
  </head>
  <body>
    <script src="./second.js"></script>
  </body>
</html>
```

Output

```
Hey i am second
Hey i am script
```

Here it has stopped the execution of script.js until the body is completely loaded. And inside the body second.js is there so that is loaded first

Tokens

The smallest unit of a programming language.

Keywords

Predefined words that have a special meaning(purpose) in JavaScript. They are always in lowercase.

Identifiers

- Name given to variables, classes, functions, or methods.

- Cannot start with a number but can contain numbers within.

- Cannot be a keyword.
- Can only contain special characters `$` and `_`.
- Should not contain spaces.

Literals

A literal is a fixed value written in a program's source code. Literals can represent different data types, such as:

- **Number literals:** `10, 200, -50`
- **String literals:** `"Hello", 'World'`
- **Boolean literals:** true , false

Variable Declarations in JavaScript

JavaScript allows declaring variables in three ways:

In JavaScript, var, let, and const are keywords used to declare variables.

Let and const are block scoped variables
They are **only accessible** inside the `{ }` block where they are defined.

Var is global scoped and function scoped variable
If declared outside a function, it's global.
If declared inside a function, it's function scoped.

1. **var**
2. **let**
3. **Const**

Keyword	**var**	**let**	**const**
Declaration	✓	✓	✗
Initialization	✓	✓	✗
Re-declaration	✓	✗	✗
Re-initialization	✓	✓	✗
Declaration + Initialization	✓	✓	✓

```
var a; // declaration
a = 10; // initialization
var a1 = 20; // dec+init
var a; // redeclaration
var a = 40; //red+reinitialization
```

```
let b; //declaration
b = 40; // init
let b1 = 30; // dec+init;
let b1; //redeclaration is not possible
```

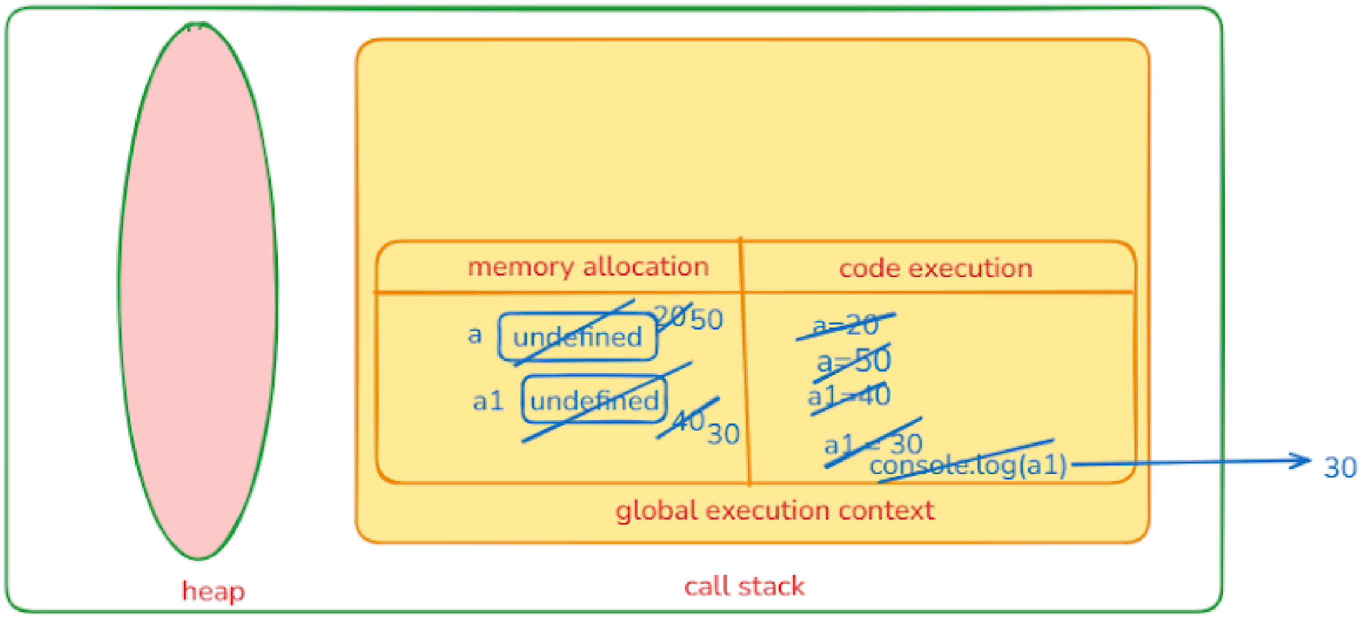
```
const c; // dec not possible
const c1 = 30; //dec+init;
const c1 ; // redec is not possible
const c1 = 30; // redec+reinit not possible
```

JS Code inside Call Stack

Js Code is executed in two phases in call stack , for the first time all the declaration happens
For the second time again from the first line the execution will happen

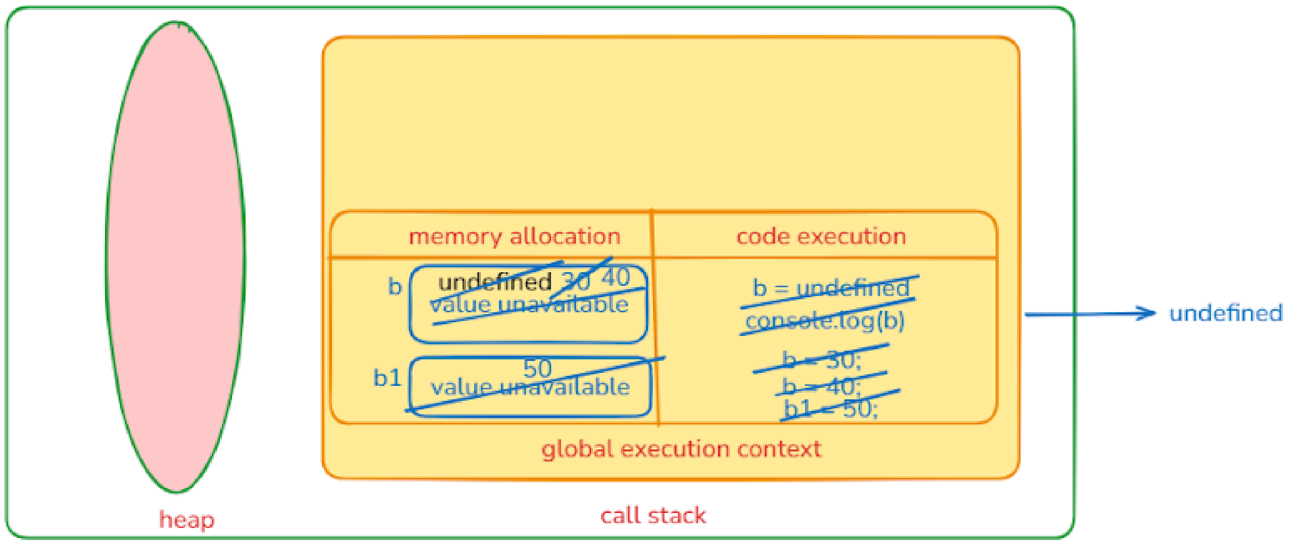
Example 1

```
var a; //dec
a = 20; //init
a = 50; // reinit
var a1 = 40; // dec+init
var a; // redec
var a1 = 30 // redec+reinit
console.log(a1);
```



Example 2

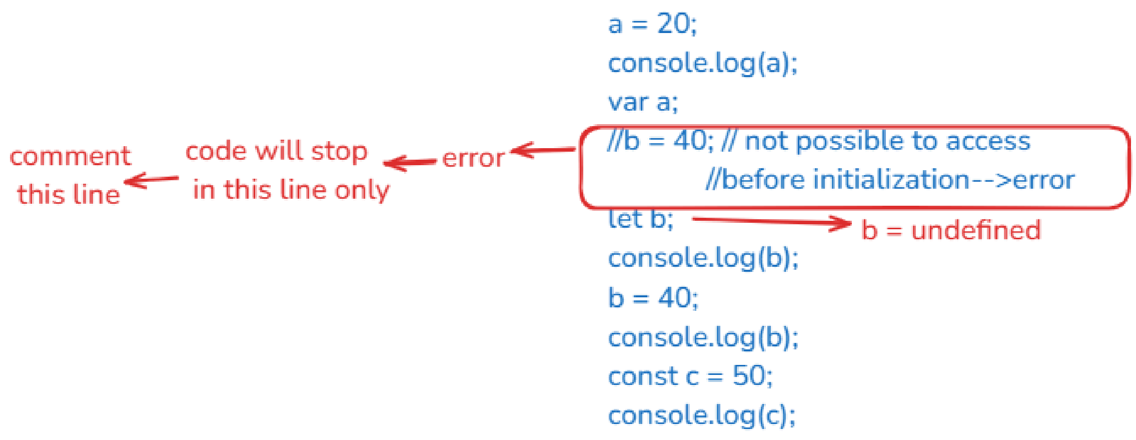
```
let b; // declaration
b = 30; // initialization
b = 40; //re-initialization
let b1 = 50; // declaration+initialization
//let b1; // re declaration is not possible
```



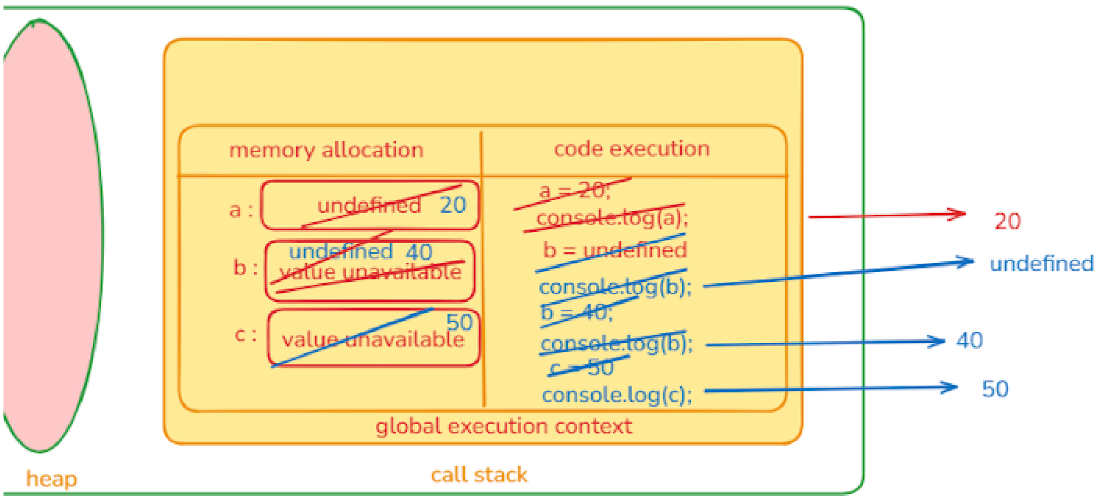
Example 3

```
a = 20;
console.log(a); //20
var a;
//b = 40
let b;
console.log(b);
b = 40;
console.log(b);
```

```
const c = 50;
console.log(c);
```



```
a = 20;
console.log(a);
var a;
//b = 40; // not possible to access
          //before initialization-->error
let b;
console.log(b);
b = 40;
console.log(b);
const c = 50;
console.log(c);
```



Hoisting

Hoisting is the process of moving variable and function declarations to the top of their respective scope before code execution.

- var can be accessed before the line declaration
- let and const are hoisted but remain in the **Temporal Dead Zone** until initialized.

\

```
//! accessing the a in line number 1 but it is declared in line 2
console.log(a); // undefined
var a = 10;
```

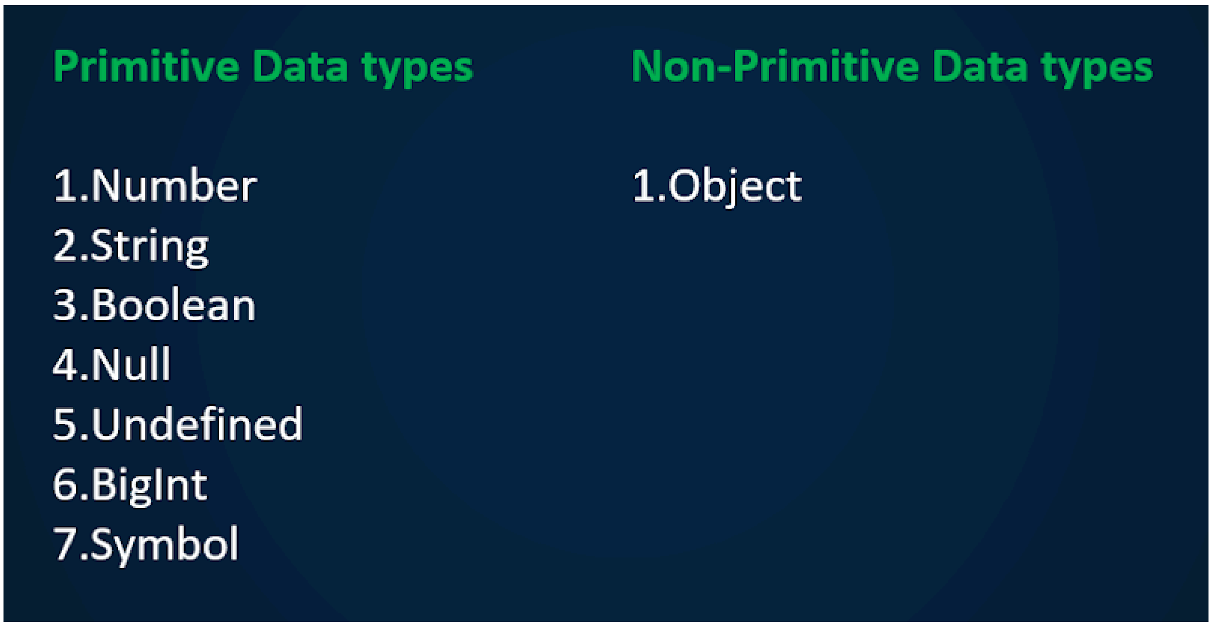
Temporal Dead Zone

JS restricts the use of variables declared with let and const before the line of declaration because they have not been initialized with any value,
If we try to access the variables declared with let and const before the line of declaration they will throw reference error

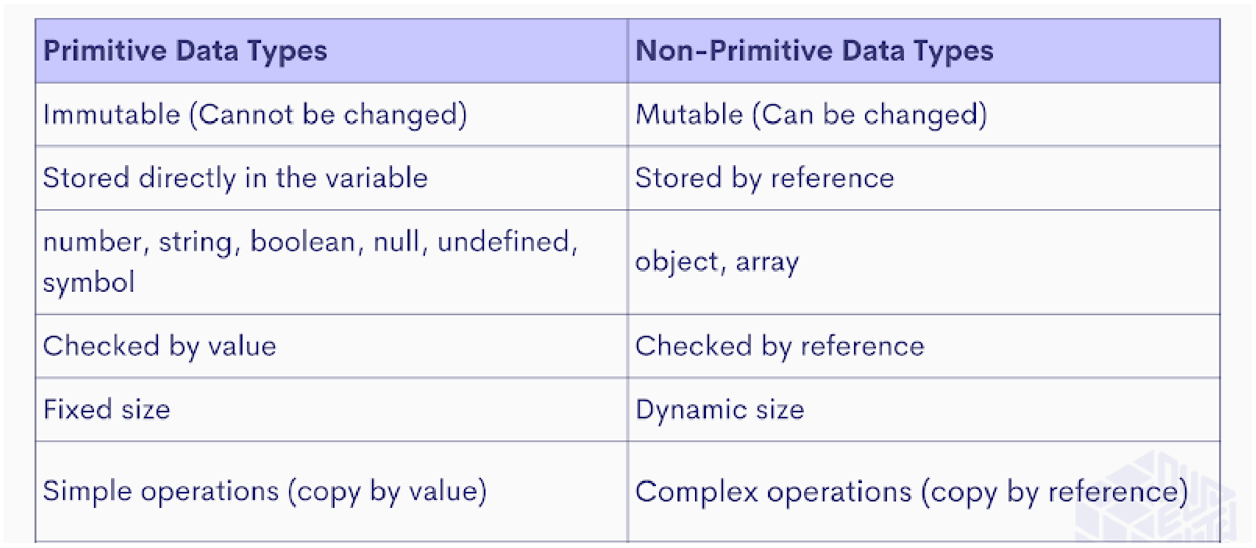
```
console.log(a); // Uncaught Reference Error Can't access a before
initialization
let a = 10;
```

Data Types

Data types specify the type of values stored in variables.



Primitive Data types	Non-Primitive Data types
1.Number	1.Object
2.String	
3.Boolean	
4.Null	
5.Undefined	
6.BigInt	
7.Symbol	



Primitive Data Types	Non-Primitive Data Types
Immutable (Cannot be changed)	Mutable (Can be changed)
Stored directly in the variable	Stored by reference
number, string, boolean, null, undefined, symbol	object, array
Checked by value	Checked by reference
Fixed size	Dynamic size
Simple operations (copy by value)	Complex operations (copy by reference)

Operators in JavaScript

Operators are special symbols that perform operations on values.

Arithmetic Operators:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)
- ** (Exponentiation)

Assignment Operators:

- = (Assign)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

Relational (Comparison) Operators:

- < (Less than)
- > (Greater than)
- <= (Less than or equal to)
- >= (Greater than or equal to)

- == (Equality check, value only)
- === (Strict equality check, value and data type both)
- != (Not equal to)

Logical Operators:

- && (AND)
- || (OR)
- ! (NOT)

Ternary Operator:

- (condition) ? true_value : false_value;
- Alternative to if-else.

thy and Falsy values

ıvaScript, a value is considered truthy if it coerces to true when evaluated in a Boolean ext . All values are truthy unless they are defined as falsy.

ıis type coercion happens in two scenarios

1.Logical operators

2.In a logical context like if else statement

Falsy values in JavaScript are:

- false
- 0 (and -0)
- 0n (BigInt zero)

- "" (empty string)
- null
- undefined
- NaN

All other values are truthy, including:

- true
- Any non-zero number
- Any non-empty string
- [] (empty array)
- {} (empty object)
- function() {} (empty function)

```
let money = 0;

if (money) {

  console.log("don't spend it all");

} else {

  console.log("you should get a job");

}
```

Type Conversion

Type conversion means converting one data type to another.

Implicit Type Conversion(Coercion) - Implicit type conversion is done by the compiler implicitly.

Eg – 5 - '3' = 2

5 + '3' = 53

- If we use numeric string with operator +, the value(operand) will be converted to string type and concatenation will happen.
- For –, *, /, and %, if operands are not numbers, they are coerced to numbers before the operation.

```
console.log("10" - 5); // Output: 5 ("10" is coerced to 10)
console.log("abc" * 2); // Output: NaN ("abc" cannot be coerced to a number)
```

- Loose Equality (==): This operator performs type coercion if the operands are of different types, attempting to convert them to a common type before comparison.

```
console.log(5 == "5"); // Output: true ("5" is coerced to 5)
console.log(0 == false); // Output: true (false is coerced to 0)
```

Explicit Type Conversion - Conversion from one data type to another explicitly it might lead to data loss

Converting to Number - Number() - Returns a number, converted from its argument
Converting to Boolean- Boolean() - Returns a boolean, converted from its argument
Converting to String- String() - Returns a number, converted from its argument

parseFloat() - Parses a string and returns a floating point number
parseInt() - Parses a string and returns an integer

== - Allows coercion
=== - Does not allow coercion
0 == '0' – True
0 == '' – True
0 === '' – False
false == '' – True

Conditional Statements

Conditional statements execute specific code blocks based on conditions.

1. if Statement

- Executes code only if the condition is true.

Syntax:

```
if (condition) {
  // Code to execute
}
```

2. if-else Statement

- Executes one block if true, another if false.

Syntax:
```
if (condition) {
   // If block code
} else {
   // Else block code
}
```

3. else if Statement

- Used for multiple conditions.

Syntax:
```
if (condition1) {
   // Code block 1
} else if (condition2) {
   // Code block 2
} else {
   // Default block
}
```

4. Nested if

- if statements inside another if.

Syntax:
```
if (condition1) {
   if (condition2) {
      // Nested if code
   }
}
```

5. switch Statement

- Matches values against cases.

Syntax:
```
switch (expression) {
   case value1:
      // Code for case1
      break;
   case value2:
      // Code for case2
      break;
   default:
      // Default case
}
```

Looping Statement

Looping statements are used to execute a block of code multiple times until a condition is met.

1. While Loop

- Executes as long as the condition is true.
- Requires initialization before the loop and updation inside the loop.

Syntax:

```
let i = 1;
while (i <= 5) {
  console.log(i);
  i++;
}
```

2. For Loop

- Used when the number of iterations is known.
- Combines initialization, condition, and updation in a single line.

Syntax:

```
for (let i = 1; i <= 5; i++) {

  console.log(i);
```

Taking User Input

```
let s = prompt("Enter a string:");
let n = Number(prompt("Enter a number:"));

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

3. do-while Loop

The do-while loop executes the block of code **at least once**, even if the condition is false. It is known as an **exit-controlled loop** because the condition is checked after executing the loop body.

Key Points:

- Executes **at least once** before checking the condition.
- Repeats as long as the condition is true.

Syntax:

```
let i = 1;  // Initialization

do {

  console.log(i);  // Code to execute

  i++;             // Updation

} while (i <= 5);  // Condition
```

Function

A **function** is a reusable block of code that performs a specific task when invoked.called.

We can invoke functions as many times as we want

Parameters and Arguments

- **Parameters**: Variables listed in the function definition.
- **Arguments**: Actual values passed to the function while invoking.

Example:

```
// Function definition with parameters 'x' and 'y'
function add(x, y) {
 return x + y;
}

// Function call with arguments 2 and 3
```

```
let result = add(2, 3);
```

Default Parameters

If a parameter is missing, it defaults to undefined, but you can set a default value.

```
function sum(x, y = 10) {
  return x + y;
}
console.log(sum(5));  // Output: 15
```

Return Statement

- Stops function execution and **returns a value**.
- If no value is returned, the function returns undefined.

Example:

```
function multiply(a, b) {
  return a * b;
}
console.log(multiply(6, 10));  // Output: 60
```

Function Invocation

- A function runs **only when it is invoked**.
- You can invoke the function as many times as you want

  ```
  myFunction(10, 2)
  ```

Example:

```
function myFunction(a, b) {
  return a * b;
}
console.log(myFunction(10, 2));  // Output: 20
```

Types Of Functions

1. **Named Function**
2. **Anonymous Function**
3. **First Class Function**
4. **Function with expression**
5. **Immediate invoke Function Expression (IIFE)**
6. **Arrow Function**
7. **Higher order Function**
8. **Callback Function**

9. **Nested Function**
10. **Generator Function**

1. Named Function:

A function declared with an identifier name
Named Functions can be hoisted (accessed before declaration)

```
// ? Syntax:
function fName(params) {
  // block of code
}
fName(args);

// ? Example:
function sum(a, b) {
  return a + b;
}
console.log(sum(10, 20)); // 30
```

2. Anonymous Function:

A function declared without an identifier.
It needs to be stored in a variable for execution.

```
// ? Syntax:
var variableName = function (params) {
  // Code
};
variableName(args);

// ? Example:
var add = function (a, b) {
  return a + b;
};
console.log(add(10, 20)); // 30
```

3. First Class Function:

A function that is stored as a value inside a variable.
It can be a named, anonymous, or arrow function.
First-class functions cannot be accessed directly without variable names.

```
var greet = function () {
  console.log("Hello, World!");
};
greet();
```

4. Function Expression:

An expression of storing any function into a variable (already covered in anonymous functions).

5. Immediately Invoked Function Expression (IIFE):
This function is executed immediately after its definition.
The first parenthesis indicates the function,
 the second indicates invocation, and the semicolon indicates the end.

```
// ? Syntax:
(function () {
  // Code
})();
```

```
// ? Example 1:
(function (a, b) {
  console.log(a + b);
})(10, 20); // 30
```

```
// ? Example 2 (with Arrow Function):
((a) => console.log(a ** 2))(10); // 100
```

6. Arrow Function:
An advanced and concise syntax for writing functions.
 It reduces code length and offers a simplified syntax.

```
// ? Syntax:
var s = (a) => a ** 2;
console.log(s(10)); // 100
```

If there is only one parameter, parentheses are optional:

```
var square = a => a ** 2;
console.log(square(10)); // 100
```

// Implicit return
In arrow functions, an implicit return allows you to return a value without using the return keyword or curly braces {}. This is useful when the function has only a single expression.

7. Higher Order Function:
A function that accepts another function as an argument or returns a function is called as a higher order function

```
function higherOrder(fn) {
  console.log("Before executing callback");
  fn();
  console.log("After executing callback");
}
```

```
higherOrder(() => console.log("Callback function executed"));
```

8. Callback Function:

A function passed as an argument to another function is called as callback function

```
function calculate(a, b, operation) {
  return operation(a, b);
}

function sum(a, b) {
  return a + b;
}

function sub(a, b) {
  return a - b;
}

function mul(a, b) {
  return a * b;
}

console.log(calculate(10, 20, sum)); // 30
console.log(calculate(10, 20, sub)); // -10
console.log(calculate(10, 20, mul)); // 200
// ? Here, 'calculate' is the Higher Order Function
// and 'sum', 'sub', 'mul' are Callback Functions.
```

9. Nested Function:

A function declared inside another function is called a Nested Function.

```
function outerFunction() {
  console.log("Outer function");

  function innerFunction() {
    console.log("Inner function");
  }

  innerFunction();
}
outerFunction();


// ! NOTE:
```

1. Closure:

When a nested function requires the variable of its parent function,
JavaScript creates closures that hold only the required variables.
Closures allow a function to access variables from its outer scope even after the outer function has finished executing.

```
function parentFunction() {
  let parentVar = "I'm a parent variable";

  function childFunction() {
    console.log(parentVar); // Accessing parent's variable
  }
  return childFunction;
}

const child = parentFunction();
child(); // "I'm a parent variable"
```

Ex 2

```
function outerFunction(x) {
 function innerFunction(y) {
   return x + y;
 }
 return innerFunction;
}

const add5 = outerFunction(5);
console.log(add5(2)); // Output: 7
```

//! Creating counter with closure

```
function counter() {
    let count = 0;
   return function getCount() {
        console.log(++count);
    }
}
let rescounter = counter()
rescounter() // 1
rescounter() //2
```

Strings

A string is a collection of characters enclosed in '', "" or ``.

```
// ! Ex:
var s1 = 'JavaScript';
var s2 = "JavaScript";
var s3 = `JavaScript`;
```

String Interpolation:

It is the process of embedding variables or solving an expression inside a string using backticks (` `).

```
// ? Ex:
var Ename = "Chetna";
var sal = 100;
console.log(`Dear ${Ename}, your salary is ${sal}`);

// ? NOTE:
// In string interpolation, we must use only backticks (` `).

// ! Length Property:
// It helps to find the length of a string.
// ? Ex:
var s = "JavaScript";
console.log(s); // "JavaScript"
console.log(s.length); // 10
```

String Methods:

```
/*
toLowerCase(), toUpperCase(), toString(), trimStart(),
trimEnd(), trim(), replace(), replaceAll(), substr(),
substring(), slice(), split(), concat(), indexOf(),
lastIndexOf(), includes(), charAt(), charCodeAt(),
String.fromCharCode()
*/
```

// ! toLowerCase():

```
// Converts all characters to lowercase.
// ? Syntax:
variableName.toLowerCase();
// ? Ex:
var s = "JavaScript";
console.log(s.toLowerCase()); // "javascript"
```

// ! **toUpperCase():**

```
// Converts all characters to uppercase.
// ? Syntax:
variableName.toUpperCase();
// ? Ex:
var s = "JavaScript";
console.log(s.toUpperCase()); // "JAVASCRIPT"
```

// ! toString():

```
// Converts a variable's data type into a string format.
// ? Syntax:
variableName.toString();
// ? Ex:
var n = 10;
```

```
console.log(n); // 10
console.log(typeof n); // "number"
var s = n.toString();
console.log(s); // "10"
console.log(typeof s); // "string"

// ! trimStart(), trimEnd(), trim():
// Removes unwanted spaces from the start, end, or both.
// ? Syntax:
variableName.trimStart(); // Removes leading spaces
variableName.trimEnd(); // Removes trailing spaces
variableName.trim(); // Removes spaces from both ends

// ? Ex:
var s = "  JavaScript  ";
console.log(s); // "  JavaScript  "
console.log(s.length); // 14
console.log(s.trimStart()); // "JavaScript  "
console.log(s.trimStart().length); // 12
console.log(s.trimEnd()); // "  JavaScript"
console.log(s.trimEnd().length); // 12
console.log(s.trim()); // "JavaScript"
console.log(s.trim().length); // 10

// ! replace():
// Replaces the first occurrence of a substring.
// ? Syntax:
variableName.replace("existingCharacter", "newCharacter");
// ? Ex:
var s = "Hii Hii Hii";
console.log(s.replace("Hii", "Bye")); // "Bye Hii Hii"

// ! replaceAll():
// Replaces all occurrences of a substring.
// ? Syntax:
variableName.replaceAll("existingCharacter", "newCharacter");
// ? Ex:
var s = "Hii Hii Hii";
console.log(s.replaceAll("Hii", "Bye")); // "Bye Bye Bye"

// ! substr(s.i , length):
// Extracts a portion of the string (deprecated in modern JavaScript).
// ? Syntax:
variableName.substr(startIndex, length);
// ? Ex:
var s = "I Love JavaScript";
console.log(s.substr(2, 4)); // "Love"
```

```
// ! substring():
// Extracts a portion of the string.
// ending index is excluded
// ? Syntax:
variableName.substring(startIndex, endIndex);
// ? Ex:
var s = "I Love JavaScript";
console.log(s.substring(2, 6)); // "Love"

// ! slice(s.i , e.i):
// Extracts a portion of the string (similar to substring).
// ending index is excluded
// here we can pass negative indexing also
// ? Syntax:
variableName.slice(startIndex, endIndex);
// ? Ex:
var s = "I Love JavaScript";
console.log(s.slice(2, 6)); // "Love"

// ? NOTE: The ending index is not included in substring() and slice().

// ! split():
// Splits a string into an array based on the (separator) argument we pass.
//the separator will not be the part of the array
// ? Syntax:
variableName.split(parameter);
// ? Ex:
var s = "I Love JavaScript";
console.log(s.split(" ")); // ["I", "Love", "JavaScript"]
var s = "I$Love$JavaScript";
console.log(s.split("$")); // ["I", "Love", "JavaScript"]

// ! concat():
// Merges two or more strings into a new string.
// ? Syntax:
firstVariable.concat(secondVariable, thirdVariable, ...);
// ? Ex:
var first = "Tata";
var second = "Consultancy";
var third = "Services";
console.log(first + " " + second + " " + third); // "Tata Consultancy Services"
console.log(first.concat(" ", second, " ", third)); // "Tata Consultancy Services"

// ! indexOf():
// Returns the index of the first occurrence of a character.
// ? Syntax:
variableName.indexOf("characterName");
```

```
// ? Ex:
var s = "JavaScript";
console.log(s.indexOf("a")); // 1

// ! lastIndexOf():
// Returns the index of the last occurrence of a character.
// ? Syntax:
variableName.lastIndexOf("characterName");
// ? Ex:
var s = "JavaScript";
console.log(s.lastIndexOf("a")); // 3

// ! includes():
// Checks if a substring exists in the string or not.
// It return a boolean value
// ? Syntax:
variableName.includes("substring");
// ? Ex:
var s = "JavaScript";
console.log(s.includes("a")); // true
console.log(s.includes("b")); // false

// ! charAt():
// Returns the character at a specific index.
// ? Syntax:
variableName.charAt(indexPosition);
// ? Ex:
var s = "JavaScript";
console.log(s.charAt(4)); // "S"

// ! charCodeAt():
// Returns the ASCII value of the character at a specific index.
// ? Syntax:
variableName.charCodeAt(indexPosition);
// ? Ex:
var s = "JavaScript";
console.log(s.charCodeAt(1)); // 97

// ! String.fromCharCode():
// Converts an ASCII code to a character.
// ? Syntax:
String.fromCharCode(ASCII_number);
// ? Ex:
console.log(String.fromCharCode(65)); // "A"
console.log(String.fromCharCode(97)); // "a"
```

Array

Arrays

An array is a collection of **homogeneous** (same type) or **heterogeneous** (different types) data elements.

Examples:

- [10, 20, 30, 40, 50](homogeneous)
- [10, "JavaScript", null, true, 50.5)](heterogeneous);

Creation of Arrays

Arrays can be created in **2 ways**:

1. **Using Literals**
2. **Using new Keyword**

1. Using Literals

Syntax: var arrayName = [element1, element2, ..., elementN];

Example:

```
var ar = [10, 20, 30, 40, 50];
console.log(ar); // [10, 20, 30, 40, 50]
```

---

2. Using new Keyword

Syntax: var arrayName = new Array();

Example:

```
var ar = new Array(); // Creates an empty array
var ar = new Array(10, 20, 30, 40, 50); // Creates an array with elements
console.log(ar); // [10, 20, 30, 40, 50]
```

Insertion in Arrays

Insertion is the process of adding elements into an array. Elements are added based on indexing.

Example:

```
var ar = []; // Empty array
console.log(ar); // []
ar[0] = 10; // Insert 10 at index 0
ar[1] = 20; // Insert 20 at index 1
ar[2] = 30; // Insert 30 at index 2
console.log(ar); // [10, 20, 30]
```

Modification in Arrays

Modification is the process of updating the elements present in an array. Elements are modified based on indexing.

Example:

```
var ar = [10, 20, 30, 40, 50];
console.log(ar); // [10, 20, 30, 40, 50]
ar[1] = 200; // Modify element at index 1
ar[2] = 300; // Modify element at index 2
ar[4] = 500; // Modify element at index 4
console.log(ar); // [10, 200, 300, 40, 500]
```

Deletion in Arrays

Deletion is the process of removing elements from an array. Elements are deleted using the delete keyword followed by the index.
Note: Deletion leaves empty places in the array.
Example:

```
var ar = [10, 20, 30, 40, 50];
console.log(ar); // [10, 20, 30, 40, 50]
delete ar[1]; // Delete element at index 1
delete ar[4]; // Delete element at index 4
console.log(ar); // [10, empty, 30, 40, empty]
console.log(ar.length); // 5 (length remains the same)
```

Advance loops on the Array

```
let nums = [30, 40, 50, 80];
```

for of loop

it will iterate through all the elements(values) of the array

```
for (let i of nums) {
  console.log(i);
}
```

for in loop

it will iterate through the index of the array

```
for (let i in nums) {
  console.log(i);
}
```

Array Methods

Mutable Methods (Modify the Original Array)

pop()

```
// Syntax: array.pop()
```

Removes the last element from an array and returns it.
Example:

```
        let arr = [1, 2, 3];
        let lastElement = arr.pop(); // Removes 3
        console.log(arr); // [1, 2] (original array is modified)
        console.log(lastElement); // 3 (removed element)
```

push()

// Syntax: array.push(element1, element2, ..., elementN)

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

```
        let arr = [1, 2];
        let newLength = arr.push(3, 4); // Adds 3 and 4
        console.log(arr); // [1, 2, 3, 4] (original array is modified)
console.log(newLength); // 4 (new length of the array)
```

shift()

// Syntax: array.shift()

Removes the first element from an array and returns it.
Example:

```
let arr = [1, 2, 3];
let firstElement = arr.shift(); // Removes 1
console.log(arr); // [2, 3] (original array is modified)
console.log(firstElement); // 1 (removed element)
```

unshift()

// Syntax: array.unshift(element1, element2, ..., elementN)

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

```
let arr = [2, 3];
let newLength = arr.unshift(0, 1); // Adds 0 and 1
console.log(arr); // [0, 1, 2, 3] (original array is modified)
console.log(newLength); // 4 (new length of the array)
```

splice()

// Syntax: array.splice(startIndex, deleteCount, newElement1, newElement2, ..., newElementN)

Adds or removes elements from an array at a specified index.
Example:

```
let arr = [1, 2, 3, 4];
let removed = arr.splice(1, 2, 'a', 'b'); // Removes 2 elements starting at index 1, adds 'a' and 'b'
console.log(arr); // [1, 'a', 'b', 4] (original array is modified)
console.log(removed); // [2, 3] (removed elements)
```

sort()

//Syntax: sort(compareFunction)

- sort method is to sort the array elements
- if we use sort method without compare function it will compare only the first digits and it won't sort properly
- we can pass a compare function to sort method

```
//ascending order
arr.sort((a, b) =< {
   return a - b;
})
//! descending order
arr.sort((a, b) =< {
   return b - a;
})
```

reverse()

// Syntax: array.reverse()

Reverses the order of elements in an array in place.

Example:

```
let arr = [1, 2, 3];
arr.reverse();
console.log(arr); // [3, 2, 1] (original array is modified)
```

copyWithin()

// Syntax: array.copyWithin(targetIndex, startIndex, endIndex)

Copies a sequence of elements within the array to a specific position.

Example:

```
let arr = [1, 2, 3, 4, 5];
arr.copyWithin(0, 3); // Copies elements from index 3 to the end and pastes them
starting at index 0
console.log(arr); // [4, 5, 3, 4, 5] (original array is modified)
```

fill()

// Syntax: array.fill(value, startIndex, endIndex)

Fills all or part of an array with a static value.

Example:

```
let arr = [1, 2, 3, 4];
arr.fill(0, 1, 3); // Fills with 0 from index 1 to 3 (exclusive)
console.log(arr); // [1, 0, 0, 4] (original array is modified)
```

Immutable Methods (Do Not Modify the Original Array)

at(index)

it will return the element present at a particular index ,
it also accepts negative index

```
let arr = [10, 20, 30, 10, 40, 50, 10];
console.log(arr.at(2)); //30
```

```
    console.log(arr.at(-2)); //40
    console.log(arr.at(-4)); //20
```

arr.indexOf(element)
it will return the index of a particular element
```
console.log(arr.indexOf(40)); //3
console.log(arr.indexOf(100)); //-1
```

lastIndexOf(element)
it will return the index of last occurence of the element
```
console.log(arr.lastIndexOf(10));
```

concat()
// Syntax: array.concat(array1, array2, ..., arrayN)
Combines two or more arrays and returns a new array.
Example:
```
let arr1 = [1, 2];
let arr2 = [3, 4];
let newArr = arr1.concat(arr2);
console.log(newArr); // [1, 2, 3, 4] (new array is created)
console.log(arr1); // [1, 2] (original array remains unchanged)
```

slice()
Syntax: array.slice(startIndex, endIndex)
Returns a copy of a portion of an array.
It will always exclude the ending index
Example:
```
let arr = [1, 2, 3, 4];
let sliced = arr.slice(1, 3); // Extracts elements from index 1 to 2
console.log(sliced); // [2, 3] (new array is created)
console.log(arr); // [1, 2, 3, 4] (original array remains unchanged)
```

join(separator)
it will join all the elements of the array and return a single string
it will return a new string
```
console.log(fruits.join(" ")); //grapes banana pineapple mango
console.log(fruits.join("$")); //grapes$banana$pineapple$mango
```

Advance Methods

//! find() , some() , every() , filter() , map() , forEach()
for all of these methods the callback function is same
callback Function -->

- in that callback function we can pass three parameters ,
- 1st parameter will be iterating variable (iterate through the elements)
- 2nd parameter will iterate through index
- 3rd parameter represents actual array

find(callback fn....)

- callback function will be executed for every element until the condition is satisfied (gets true).
- it will return that first element which satisfies the condition

```
let nums = [10 , 8 , 16 , 6 , 9 , 4 , 15];
let res = nums.find((el , i , ar)=>{
   console.log("iteration : " , i+1)
   return el>10;
   // console.log(el , i)
})
console.log(res)
```

some(callback fn...)

- callback function will be executed for every element until the condition gets true.
- same is same as find only difference is it will return boolean values
- some will return true if any one element satisfies the condition

```
let nums = [10 , 8 , 16 , 6 , 9 , 4 , 15];
let res = nums.some((el , i , ar)=>{
   console.log("iteration : " , i+1)
   return el<16;
})
console.log(res) // false
```

every(callback fn...)

- every will return true if all the elements satisfies the condition
- if any element is returning false for the condition there only it will come out of the loop
- It will return boolean value

```
let nums = [10, 8, 16, 6, 9, 4, 15];
let res = nums.every((el, i, ar) => {
  console.log("iteration : ", i + 1);
  return el <= 16;
});
console.log(res); // true
```

forEach(callback fn..)

it won't return anything it is just used to iterate only

```
let arr = [10, 21, 40, 90, 13];
arr.forEach((el, i, arr) => {
  console.log(el, i);
});
```

filter(callback fn...)

- Filter method is used to filter out the array based on some condition returned from callback function
 It will execute the callback function for every element of the array

- it will return a new array , in that array all the elements satisfies the condition will be there
- It does not change the original array

```
let arr = [10, 21, 40, 90, 13];
let even = arr.filter((el, i) => {
  return el % 2 === 0;
  return false;
});
let odd = arr.filter((el, i) => el % 2 !== 0);
console.log(even);
console.log(odd);

//! Assignment
let names = ["varuni", "akshay", "aakash", "pranav", "aatul", "shaubham"];
//! filter the names with exactly 6 characters
//! filter the names with exactly 3 vowel
//! filter the elements starting with vowel
```

map(callback Fn...)

- Map method is used to update the elements of the array based on condition returned from callback function
- It will execute the callback function for every element of the array
- it will return a new array with updated elements.
- It does not change the original array

```
let arr = [1, 2, 3, 4];
let newArr = arr.map((el, ind, ar) => {
   return el * 2;
});
console.log(newArr); // [2, 4, 6, 8]
```

reduce(callback Fn...)

- reduce method is used to reduce the array to a single value by executing a callback function for each element (left to right)
- we have to return the accumulated value from the callback function of the reduce. The actual array will not be modified
- reduce will accept a callback function, that callback function can have 4 parameters
- 1st parameter is the accumulator, which accumulates the returned values
- 2nd parameter is the iteration variable that will iterate through the values of an array
- 3rd parameter is for the indexes
- 4th parameter will represent the actual array
- reduce also accepts an optional initial value as the second argument

```
// Example
let arr = [1, 2, 3, 4];
let sum = arr.reduce((acc, el, ind, ar) => {
   return acc + el;
}, 0);
console.log(sum); // 10
```

```
    return el % 2 === 0;
});
console.log(elEven); // true
```

reduceRight(callback Fn...)

```
// reduceRight method is similar to reduce, but it processes the array from right to left
        let arr = [1, 2, 3, 4];
        let result = arr.reduceRight((acc, el, ind, ar) => {
            return acc - el;
        });
        console.log(result); // -2 (4 - 3 - 2 - 1)
```

Summary Table

Method	**Purpose**	**Callback Arguments**	**Returns**
filter()	Filters elements based on a condition	(element, index, array)	New array
map()	Transforms each element	(element, index, array)	New array
reduce()	Reduces array to a single value (left-right)	(accumulator, element, index, array)	Single value
some()	Checks if at least one element passes a test	(element, index, array)	Boolean
every()	Checks if all elements pass a test	(element, index, array)	Boolean
find()	Finds the first element that passes a test	(element, index, array)	Single element or undefined
reduceRight()	Reduces array to a single value (right-left)	(accumulator, element, index, array)	Single value

What is an Object?

- An object is a **collection of key-value pairs** (properties).
- It is used to represent real-world entities in a program.
- Keys and value pair is called **property**, and values can be anything like strings, numbers, booleans, functions, etc.

Creating an Object

- **Object Literal Syntax (Most Common)**

```
let person = {
  name: "Rohit", → key and value are separated by colon (:)
  age: 37,
  team: "MI",
};
```

Rules for Keys:

- Keys can be **strings**, **numbers**, or **symbols**.
- If a key doesn't follow the rules of an identifier (e.g., starts with a number or has special characters), it must be wrapped in **quotes**.
- If you use the same key more than once, the last value will overwrite the previous one.

```
let marker = {
  color: "Black",
  isMarried: true,
  77: "Dhoni", // Number as a key
  "leng*h": 10, // Key with special characters
  "1marker": "hey", // Key starting with a number
};
```

Accessing Object Properties

1. **Dot Notation (.):**
   - **Use the property name directly.**
   - **Example: student.name → "John".**
2. **Bracket Notation ([ ]):**
   - **Use when the property name is dynamic or stored in a variable.**
   - **Example: student["name"] → "John".**

Adding/Removing Properties

1. **Add:**
   - **Use dot or bracket notation.**
   - **Example: student.age = 20 or student["age"] = 20.**
2. **Remove:**

- ○ **Use delete.**
- ○ **Example: delete student.age.**

Object Methods

1. **hasOwnProperty:**
   - ○ **Check if a property exists.**
   - ○ **Example: student.hasOwnProperty("name") → true.**
2. **Object.keys:**
   - ○ **Get all keys in an array.**
   - ○ **Example: Object.keys(student) → ["name", "age"].**
3. **Object.values:**
   - ○ **Get all values in an array.**
   - ○ **Example: Object.values(student) → ["John", 20].**
4. **Object.entries:**
   - ○ **Get key-value pairs as nested array.**
   - ○ **Example: Object.entries(student) → [["name", "John"], ["age", 20]].**

5.Object.seal

Prevents adding or deleting properties but allows updating existing ones.

```
const person = { name: "Alice", age: 30 };
    Object.seal(person);

    person.age = 31; // Allowed
    person.gender = "female"; // Not allowed
    delete person.name; // Not allowed
```

6.Object.freeze

Prevents adding, deleting, or updating properties.

```
const car = { brand: "Toyota", model: "Camry" };
    Object.freeze(car);

    car.model = "Corolla"; // Not allowed
    car.year = 2020; // Not allowed
    delete car.brand; // Not allowed
```

7.Object.isSealed()

The Object.isSealed() method checks if an object is sealed. It returns true if the object is sealed and false otherwise.

8.Object.isFrozen()

The Object.isFrozen() method determines if an object is frozen. It returns true if the object is frozen and false otherwise.

Ways to Create an Object in JavaScript

1. Object Literal

The simplest way to create an object using curly braces {}:

```
let obj = {
  name: "Peter Parker",
  age: 17,
  address: "New York",
};
```

---

2. Object Constructor

Creates an empty object; properties can be added later.

```
let obj = new Object();

obj.age = 20;

console.log(obj); // {age: 20}

let obj2 = new Object({ age: 20, name: "Ram" });

console.log(obj2); // {age: 20, name: 'Ram'}
```

---

3. Constructor Function

Used to create multiple instances with similar properties.

```
function Student(name, age) {
  this.name = name;
  this.age = age;
}
let st1 = new Student("Pranav", 20);
console.log(st1); // Student {name: 'Pranav', age: 20}
```

How new Works:

1. Creates an empty object {}.
2. Assigns this to the new object.
3. Links the object's prototype.
4. Assigns passed arguments as properties.
5. Returns the new object.

Using Prototype for Shared Methods

```
function People(name, age) {
  this.name = name;
  this.age = age;
}
People.prototype.addAge = function () {
  console.log(this.age + 100);
};
let person1 = new People("Pranav", 19);
person1.addAge(); // 119
```

---

4. Class (ES6)

A cleaner way to define object blueprints, works similarly to constructor functions.

```
class Person {
  constructor(name, age) {

    this.name = name;

    this.age = age;

  }

  addAge() {

    console.log(this.age + 100);

  }

}

let person1 = new Person("Pranav", 19);

console.log(person1);
```

5. Object.assign()

Copies properties from one or more objects into a target object.

```
let obj1 = { name: "Pranav" };

let obj2 = { character: "sweet" };

console.log(Object.assign({}, obj1, obj2)); // { name: 'Pranav', character: 'sweet' }
```

---

6. Object.create()

Creates a new object with a specified prototype.

```
let parentObj = {

  name: "Atul",
  printName() {

    console.log(this.name);

  },

};

let obj = Object.create(parentObj);

console.log(obj.name); // "Atul" (inherited)

obj.printName();
```

This Keyword

In JavaScript, the this keyword refers to the object that is executing the current function. Its value depends on how a function is called, not where it is defined. Let’s break it down simply

```
//! 1st globally
console.log(this); // Window object
//! 2nd in a function
function show() {
  console.log(this);
}
show(); // In browser: Window
//! 3rd inside a method
const person = {
  name: "Akshay",
  greet: function () {
    console.log(this.name);
  }
};
person.greet(); // "Akshay"
//! 4th constructor function
function Person(name) {
  this.name = name;
}
const p = new Person("Akshay");
console.log(p.name); // "Akshay"
//! 5th event listener
document.querySelector("button").addEventListener("click", function () {
  console.log(this); // the button element
});
//! 6th Inside a class
```

```
class User {
  constructor(name) {
    this.name = name;
  }
  showName() {
    console.log(this.name);
  }
}
const u = new User("Akshay");
u.showName(); // "Akshay"
```

Context	this refers to
	Global object (window)
ıod)	Global object / undefined (strict)
	That object
	Lexical (surrounding) this
	New instance being created
	Class instance
ılar function)	DOM element
w function)	Lexical this

Call ,Apply , Bind

In JavaScript, functions are objects that come with methods like call(), apply(), and bind() to explicitly set the this context, allowing for flexible function invocation patterns.

1. call() Method

The call() method invokes a function immediately, setting its this context to a specified object and passing arguments individually.

Syntax :

```
functionName.call(thisArg, arg1, arg2, ...)

function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}.`);
}

const person = { name: "Alice" };

introduce.call(person, "Hello"); // Output: Hello, I'm Alice.
```

In this example, introduce.call(person, 'Hello') sets this to the person object and passes 'Hello' as an argument to the introduce function.

2. apply() Method

The apply() method is similar to call(), but it accepts arguments as an array or array-like object.

Syntax:

```
functionName.apply(thisArg, [arg1, arg2, ...])
```

```
function introduce(greeting, punctuation) {

  console.log(`${greeting}, I'm ${this.name}${punctuation}`);

}

const person = { name: "Bob" };

introduce.apply(person, ["Hi", "!"]); // Output: Hi, I'm Bob!
```

Here, introduce.apply(person, ['Hi', '!']) sets this to the person object and passes 'Hi' and '!' as arguments to the introduce function.

3. bind() Method

The bind() method creates a new function that, when invoked, has its this context set to a specified object, with optional arguments pre-filled. This new function can be called later.

Syntax :

```
const boundFunction = functionName.bind(thisArg, arg1, arg2, ...)
```

```
function introduce() {

  console.log(`I'm ${this.name}.`);

}
```

Example

```
const person = { name: "Charlie" };

const boundIntroduce = introduce.bind(person);

boundIntroduce(); // Output: I'm Charlie.
```

Math Object

Properties:

- **Math.PI**: Represents the ratio of the circumference of a circle to its diameter, approximately 3.14159.
- **Math.SQRT1_2**: Represents the square root of 1/2, approximately 0.707.
- **Math.SQRT2**: Represents the square root of 2, approximately 1.414.

Methods:

- **Math.abs(x)**: Returns the absolute value of x. For example, Math.abs(-90) returns 90.
- **Math.floor(x)**: Rounds x down to the nearest integer. For example, Math.floor(16.9) returns 16.
- **Math.ceil(x)**: Rounds x up to the nearest integer. For example, Math.ceil(16.1) returns 17.
- **Math.round(x)**: Rounds x to the nearest integer. For example, Math.round(16.5) returns 17.
- **Math.trunc(x)**: Removes the fractional part of x, effectively truncating it. For example, Math.trunc(16.7) returns 16.
- **Math.random()**: Generates a pseudo-random number between 0 (inclusive) and 1 (exclusive). For example, Math.random() might return 0.123456.

```
function getOtp(){
let otp = Math.floor(Math.random()*8999)+1000;

return `Your OTP is ${otp}`;

}

console.log(getOtp())
```

! Assignment --> Guessing Game , OTP Mathcing

Generating Random Numbers in a Specific Range:

To generate a random integer between two values, you can use the following formula:

```
let min = 1;

let max = 10;

let randomValue = Math.floor(Math.random() * (max - min + 1)) + min;

console.log(randomValue); // Outputs a random integer between 1 and 10
```

Date Object:

The Date object in JavaScript is used to work with dates and times. You can create a new date object using new Date(), which returns the current date and time.

Retrieving Date Components:

- **getFullYear()**: Returns the four-digit year (e.g., 2025).
- **getMonth()**: Returns the month (0-11). January is 0, February is 1, and so on.
- **getDate()**: Returns the day of the month (1-31).

- **getDay()**: Returns the day of the week (0-6). Sunday is 0, Monday is 1, and so on.
- **getHours()**: Returns the hour (0-23).
- **getMinutes()**: Returns the minutes (0-59).
- **getSeconds()**: Returns the seconds (0-59).
- **getMilliseconds()**: Returns the milliseconds (0-999).

Setting Date Components:

You can modify specific components of a date using setter methods:

```
let purchaseDate = new Date();

purchaseDate.setFullYear(2020);

purchaseDate.setMonth(8); // September (months are 0-indexed)

purchaseDate.setDate(15);

purchaseDate.setHours(13);

purchaseDate.setMinutes(30);

console.log(purchaseDate); // Outputs: Tue Sep 15 2020 13:30:00
```

BOM(Browser Object Model)

BOM allows JavaScript to interact with the browser.
It controls things like window size, location, and history.
The **window** object is the main entry point for accessing BOM features.
BOM helps manipulate the browser environment outside the webpage.

Window Properties:

1. document
2. screen
3. history
4. navigator
5. location
6. innerheight

7. outerheight
8. innerwidth
9. outerwidth

1.Document:
It always represents the current html document. By using this, we can make changes in the HTML document.

2. Screen:
It provides the basic information about the screen like height , width , orientation , etc...

3. History:
It will keep a track of web pages you opened and it will store as history. We can perform forward() , back() operations using this.

4. Navigator:
It provides information about browser like appname , aversion , etc...

5. Location:
It provides the basic information about the current location like path , hostname , href , etc...

6. InnerHeight:
It calculates the browser height excluding console area.

7. OuterHeight:
It calculates the browser height including console area.

8. Innerwidth:
It calculates the browser width excluding console area.

9. Outerwidth:
It calculates the browser width including console area.

Window methods:

1. prompt()
2. alert()
3. confirm()
4. open()
5. close()
6. setTimeOut()
7. setInterval()
8. clearTimeOut()
9. cleatInterval()

1. **prompt():**
It helps to take user input. It will display a popup message in the browser.

```
var userInput = prompt("Please enter your name:");
```

2. **alert():**
It will display a popup message with a button "OK".

```
alert("This is an alert!");
```

3. **confirm():**
It will display a popup message with "OK" & "CANCEL" buttons.

```
var result = confirm("Do you want to continue?");
```

4. **open():**
It helps to open a new webpage in a new tab.

```
window.open("https://www.example.com", "_blank");
```

5. **close():**
It allows us to close the current webpage.

```
 window.close();
```

6. **setTimeout():**
It holds the execution until the completion of time.
Syntax: setTimeout(callback_fn, timeout);

```
setTimeout(() => alert("This message will appear after 3 seconds"), 3000);
```

7. **setInterval():**
It will perform the same task multiple times with the time gap of interval time.
Syntax: setInterval(callback_fn, intervalTime);

```
setInterval(() => alert("This will show every 5 seconds"), 5000);
```

8. **clearTimeout():**
It will stop the working of the setTimeout() method.

```
var timeoutId = setTimeout(() => alert("This won't show"), 5000);
```

```
clearTimeout(timeoutId); // This will cancel the timeout
```

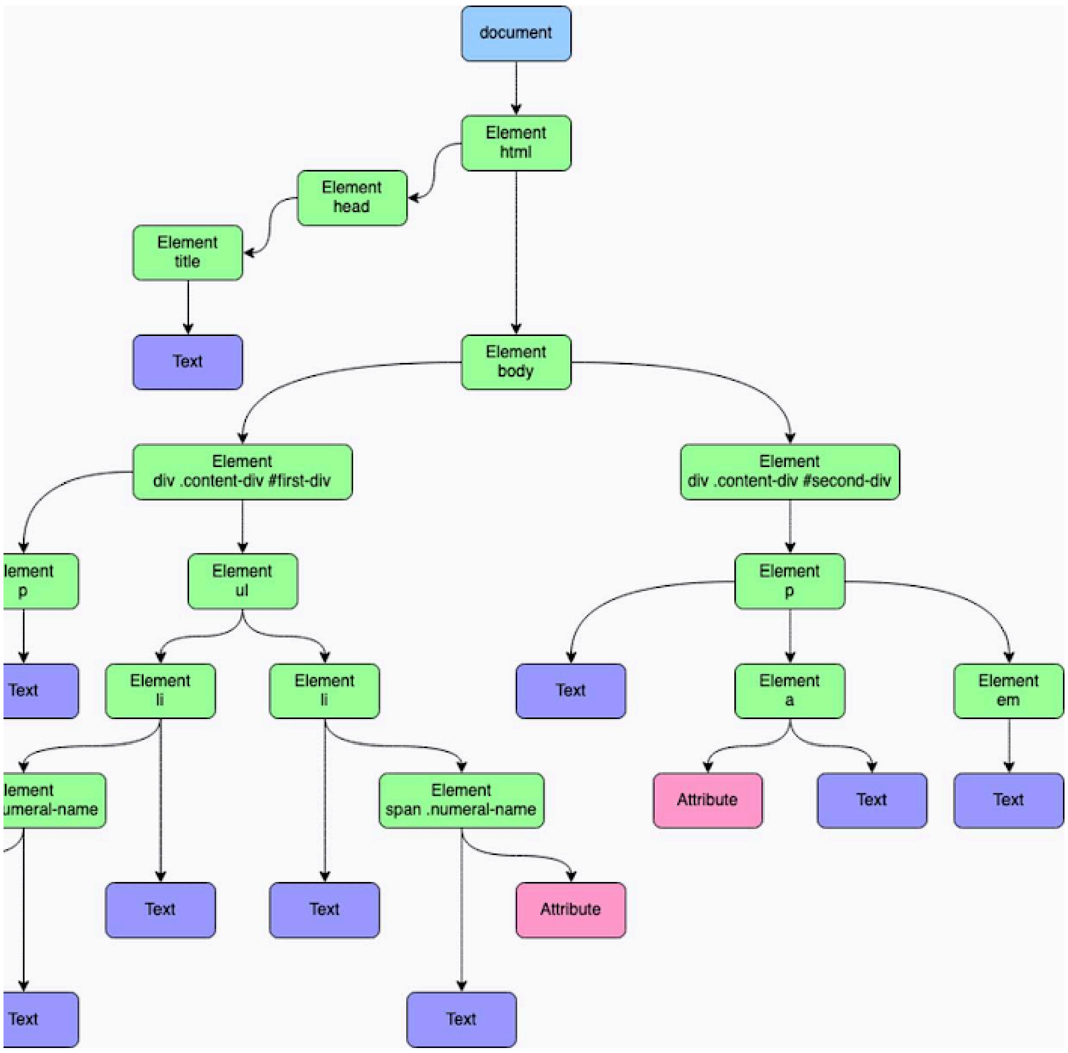
9. **clearInterval():**
 It will stop the working of the setInterval() method.

```
var intervalId = setInterval(() => alert("This won't show anymore"), 5000);

clearInterval(intervalId); // This will cancel the interval
```

DOM (Document Object Model)

- DOM stands for Document Object Model
- DOM is the part of the browser (Its a browser API), which is used to manipulate the html elements , and styles
- DOM is a tree like hierarchical structure representing the html document
- It is created by the browser , because javascript directly cannot communicate with the html elements , so browser creates the dom where each and every html element is converted to a node (object format)
- Each **HTML element** becomes an **element node**, attributes are **attribute nodes**, text content forms **text nodes**, and comments are **comment nodes**.



DOM Selectors

DOM selectors are used to target and access HTML elements in a document.

1. getElementById()
- Selects an element by its ID.
- Returns a single element.
- Syntax: document.getElementById("idName");

2. getElementsByClassName()
- Selects elements by class name.
- Returns a collection (HTMLCollection) of matching elements.
- Syntax: document.getElementsByClassName("className");

3. getElementsByTagName()
- Selects elements by tag name (e.g., div, p, h1).
- Returns a collection (HTMLCollection) of matching elements.
- Syntax: document.getElementsByTagName("tagName");

4. getElementsByName()
- Selects elements by their name attribute.
- Returns a collection (NodeList) of matching elements.
- Syntax: document.getElementsByName("name");

5. querySelector()
- Selects the first matching element based on an ID (#id), class (.class), or tag (tag).
- Syntax: document.querySelector("#idName / .className / tagName");

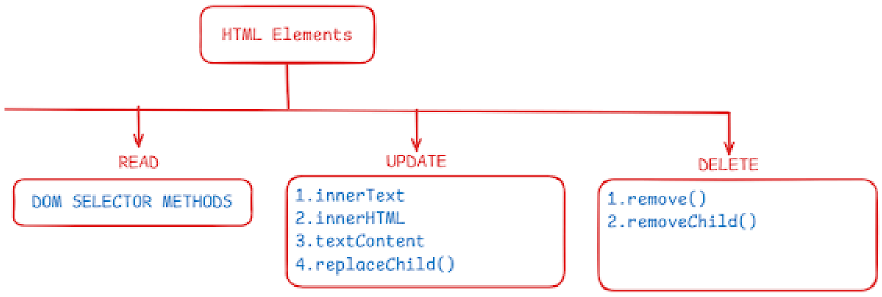
6. querySelectorAll()
- Selects all matching elements based on an ID, class, or tag.
- Returns a NodeList (which is not an array but can be looped through).
- Syntax: document.querySelectorAll("#idName / .className / tagName");

Note:
- getElementsByClassName(), getElementsByTagName(), and getElementsByName() return collections, not actual arrays.
- These collections support indexing and iteration but do not support array methods like push(), pop(), shift(), etc.

CRUD Operations

We can perform CRUD Operations on HTML Elements , HTML attributes and inline style

CRUD ON HTML Elements



eateElement('tag')

Creates a new HTML element.

```
    let variableName = document.createElement("elementName");
  let div = document.createElement("div");
```

2. append()

Appends content (text or nodes) to a parent element.

```
  parent.append(content or node);
  section.append("This is Text Information");
    section.append(div);
```

3. appendChild()

Appends only **nodes** to a parent element.

```
  parent.appendChild(node);

 section.appendChild("This is Text Information"); //  Error (Text must be a node)

    section.appendChild(div); // ✅ Works
```

4. textContent, innerText, innerHTML

Used to update the content inside an element.

- textContent and innerText display text.
- innerHTML allows adding HTML tags.

```
  selector.textContent = "content";
    selector.innerText = "content";
    selector.innerHTML = "<b>This is content</b>"; // Allows HTML

 div.textContent = "This is content";
    div.innerText = "This is content";
    div.innerHTML = "<b>This is content</b>";
```

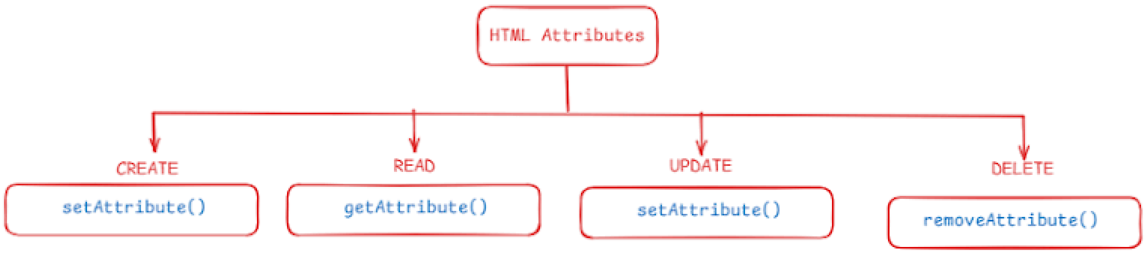
5. replaceChild()
Replaces an existing node with a new node.

```
parent.replaceChild(newNode, oldNode);
section.replaceChild(h1, div);
```

6. removeChild()
Removes a child node from a parent element.

```
parent.removeChild(node);
section.removeChild(h1);
```

CRUD ON HTML Attributes



1. setAttribute():
It will add a new attribute to the element.
If the attribute is already present it will update the attribute value.

```
//? Syntax:
selector.setAttribute("attributeName","attributeValue")
//? Ex:
div.setAttribute("id","demo");
```

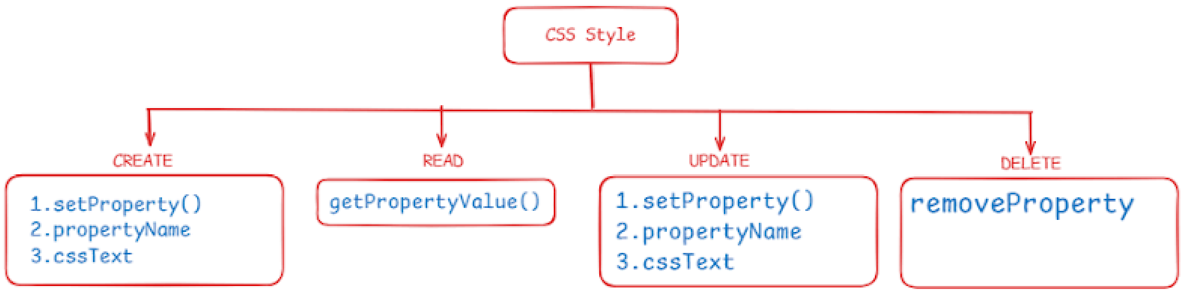
2. getAttribute():
It will return the attribute value of an element.

```
//? Syntax:
selector.getAttribute("attributeName")
//? Ex:
div.getAttribute("id"); // demo
```

3. removeAttribute():
It will remove the existing attribute of an element.

```
//? Syntax:
selector.removeAttribute("attributeName")
//? Ex:
div.removeAttribute("id");
```

CRUD ON CSS Styles



1. setProperty(), propertyName, cssText

These methods are used to add CSS properties to an element:

- setProperty() and propertyName allow adding one property at a time.
- cssText allows adding multiple properties at once, but it overwrites existing styles.

```
selector.style.setProperty("property-name", "property-value");
selector.style.propertyName = "propertyValue";
selector.style.cssText = "property1: value1; property2: value2; ...";

div.style.setProperty("font-size", "40px"); // Using setProperty()
div.style.fontSize = "40px"; // Using propertyName
div.style.cssText = "font-size: 40px; color: red;"; // Using cssText
```

2. getPropertyValue()

Used to retrieve the value of a CSS property from an element.

```
selector.style.getPropertyValue("property-name");

div.style.getPropertyValue("font-size"); // Returns "40px"
```

3. removeProperty()

Used to remove a specific CSS property from an element.

```
selector.style.removeProperty("property-name");
div.style.removeProperty("font-size"); // Removes font-size property
```

Events in JavaScript

Events are actions triggered by the browser or user (e.g., clicking a button, pressing a key, resizing a window, etc.).

Mouse Events	Key Events	Form Events	Window Events
click dblclick mouseup mousedown mouseover mouseout mousemove contentmenu	keypress keydown keyup	submit reset focus blur change	load resize scroll

Ways to Attach Events in JavaScript

We can attach events to an HTML element in three ways:

1. HTMLAttributes
2. DOMProperties
3. addEventListiners

1. Using HTML Attributes

Here, the event is assigned directly as an attribute in the HTML tag.

```
<tagname onEventName="JavaScript Code">Content</tagname>;
        <div onclick="alert('Click Event is triggered')">Click Me</div>;
```

2. Using DOM Properties

Here, we attach events using **JavaScript properties**.

```
element.onEventName = function;

let div = document.getElementById("myDiv");
div.onclick = function () {
   div.style.cssText = "color: white; background-color: teal; border: 1px solid; text-align: center;";
};
```

3. Using addEventListener()

This is the best and most flexible way to attach events.

- We can attach multiple functions to the same event.
- We can remove the event later using removeEventListener().
- It supports event propagation.

```
element.addEventListener("eventName", function, useCapture);

let button = document.getElementById("myButton");

  button.addEventListener("click", () => {
    console.log("Click Event is Triggered");
  });
```

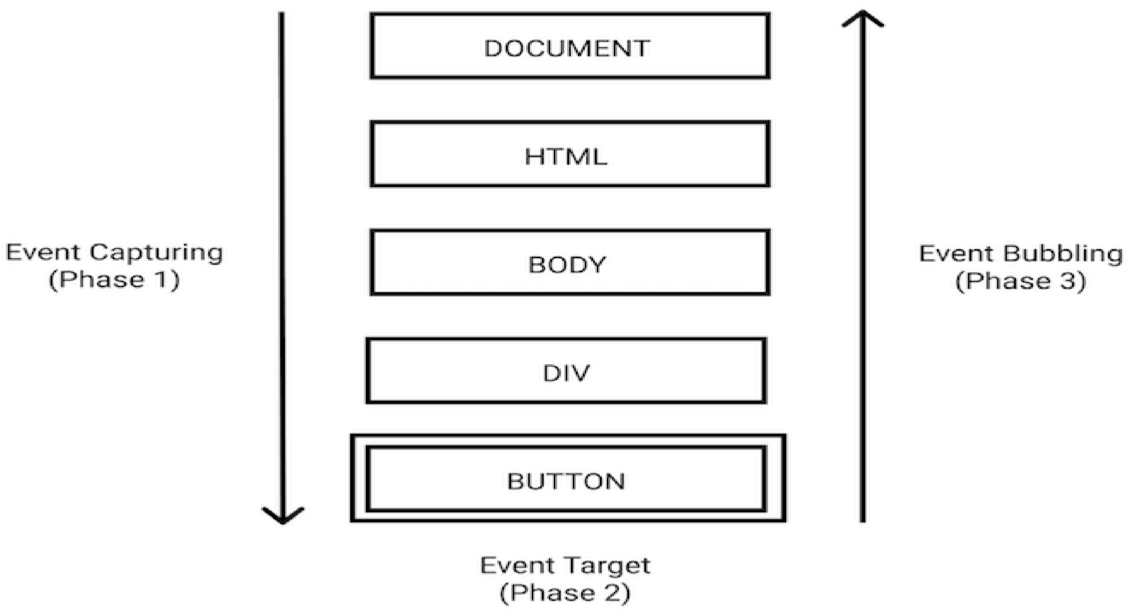
Understanding the Third Parameter in addEventListener()

The third parameter (useCapture) is optional and controls event propagation:

- true → Capturing phase (event runs from parent to child)
- false → Bubbling phase (event runs from child to parent, default behavior)

Event Propagation

Event propagation refers to how events flow through the DOM hierarchy when an event occurs. It consists of three phases:



1. Capturing Phase (Event Capturing/Trickling)
   - The event starts from the window and travels down to the target element.
2. Target Phase (Event Execution)
   - The event reaches the target element and executes its handler.
3. Bubbling Phase (Event Bubbling)
   - The event bubbles up from the target element to the window.

Example

```
parent.addEventListener( "click", () => {
    console.log("Parent Clicked");
  },  true ); // Capturing mode

 child.addEventListener("click",() => {
    console.log("Child Clicked");
  },true ); // Capturing mode

 parent.addEventListener( "click", () => {
    console.log("Parent Clicked");
  },  false); // bubbling mode

 child.addEventListener("click",() => {
    console.log("Child Clicked");
  },false ); // bubbling mode
```

Event Object

The event object (event or e) is an automatically passed argument to the event handler function, when an event occurs in JavaScript. It contains details about the event, such as the type of event, the target element, mouse coordinates, key presses, and more.

```
document.getElementById("btn").addEventListener("click", function (e) {
  console.log(e); // Logs the event object to the console
});
```

e.preventDefault()
You use preventDefault() when you want to stop a default browser behavior, such as:

1. **Stopping form submission**
2. **Preventing link navigation**
3. **Disabling right-click menu**

Stopping form submission

```
document.getElementById("myForm").addEventListener("submit", function (e) {
  e.preventDefault(); // Stops the form from reloading the page
  alert("Form submission prevented!");
});
```

- **Without preventDefault(): The form will submit and reload the page.**
- **With preventDefault(): The form won't submit, allowing JavaScript validation.**

Promises(ES6)

A promise in JavaScript is an object representing the eventual result(completion or failure) of an asynchronous operation.

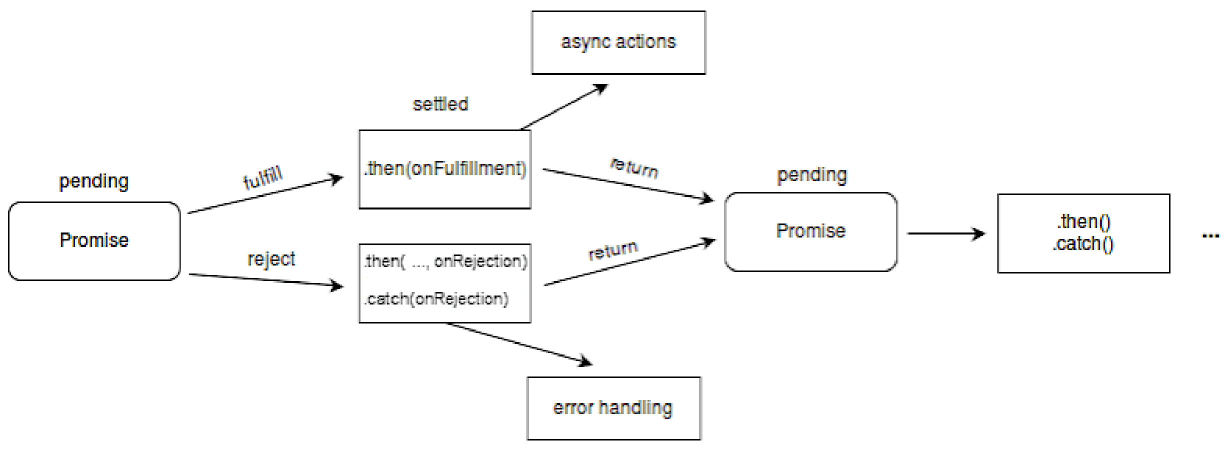
It acts as a placeholder for a value that may not be immediately available. Promises help manage asynchronous code

```
let myPromise = new Promise((resolve, reject) => {
  let success = true; // Simulating success or failure
  setTimeout(() => {
    if (success) {
      resolve("Promise Resolved Successfully!");
    } else {
      reject("Promise Rejected!");
    }
  }, 2000);
});
```

State Of Promise

A promise is in One Of these States:

- **pending:** initial state, neither fulfilled nor rejected.
- **fulfilled:** meaning that the operation was completed successfully.
- **rejected:** meaning that the operation failed.



Handling a Promise

#To handle promises we have some instance methods

- **then() :** handles resolved state
- **catch() :** handles rejected state
- **finally() :** will be executed in either resolved or rejected state

```
myPromise

 .then(message => console.log("Success:", message))

 .catch(error => console.log("Error:", error))

 .finally(() => console.log("Promise Completed!"));
```

Chaining Promises

Each .then() returns a new Promise, allowing chaining.

If we return anything from then block it will be wrapped again in a promise so to handle that we have to use one more then block .

```
let chainPromise = new Promise(resolve => {

  setTimeout(() => resolve(10), 1000);

});




chainPromise.then(num => {

      console.log(num); // 10

      return num * 2;

    })

    .then(num => {

      console.log(num); // 20

      return num * 3;

    })

    .then(num => {

      console.log(num); // 60

    });
```

Async and await

The async and await keywords in JavaScript make working with Promises much simpler and more readable. Instead of chaining .then() and .catch(),
we can use await inside an async function to pause execution until a Promise is resolved or rejected.

async keyword

The async keyword is used before a function declaration to **make it return a Promise**.

- Async keywords will be used before the function.
- Async will make the function asynchronous.
- Async will always return a promise.
- If a function returns non-promise data,it wraps into a promise and it will return it

await keyword

The await keyword is used **inside an async function** to wait for a **Promise** to resolve before continuing execution.

```
async function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject("Error: Something went wrong!"), 2000);
  });
}

async function getData() {
  try {
    let data = await fetchData();
    console.log(data);
  } catch (error) {
    console.log(error); // Output: Error: Something went wrong!
  }
}
getData();
```

JSON

- JSON stands for **JavaScript Object Notation**.
- JSON is very lightweight to store and interchange data.
- JSON stores data in the form of a **string**.(key should be in double quotes always)
- JSON is a format for storing and transporting data.
- JSON is **language-independent**.
- JSON is often used when data is sent from a server to a webpage.

Data Types Allowed in JSON:

- Object
- Boolean
- Number
- String
- Array
- Null

Example of JSON data with all possible data types

```
{
  "string": "Hello, World!",
  "number": 42,
  "boolean": true,
  "nullValue": null,
  "array": [1, 2, 3, "apple", false],
  "object": {
    "nestedString": "Nested value",
    "nestedNumber": 99
  }
}
```

JavaScript Object vs. JSON Object

JavaScript Object

- In JavaScript objects, it is not required to enclose keys in double quotes ("").
- It accepts all data types.
- It does not store data in the form of a string.

JSON Object

- In JSON, keys must be enclosed in double quotes ("").

- It does not accept data types like BigInt, Function, Date, etc.

JSON Methods

1. JSON.stringify()

Converts JavaScript data into JSON format.

```
let jsonData =
'{"emp":[{"pname":"watch","prodPrice":"3000"},{"pname":"phone","prodPrice":"15000"}]}';
let parsedData = JSON.parse(jsonData);
console.log(parsedData);
```

2. JSON.parse()

Converts JSON data into JS format.

```
let obj = {
  emp: [
    { pname: "watch", prodPrice: "3000" },
    { pname: "phone", prodPrice: "15000" },
  ],
};
let jsonData = JSON.stringify(obj);
```

AJAX

AJAX (Asynchronous JavaScript and XML) is a web development technique that enables web pages to update content dynamically without requiring a full page reload.

What is AJAX?

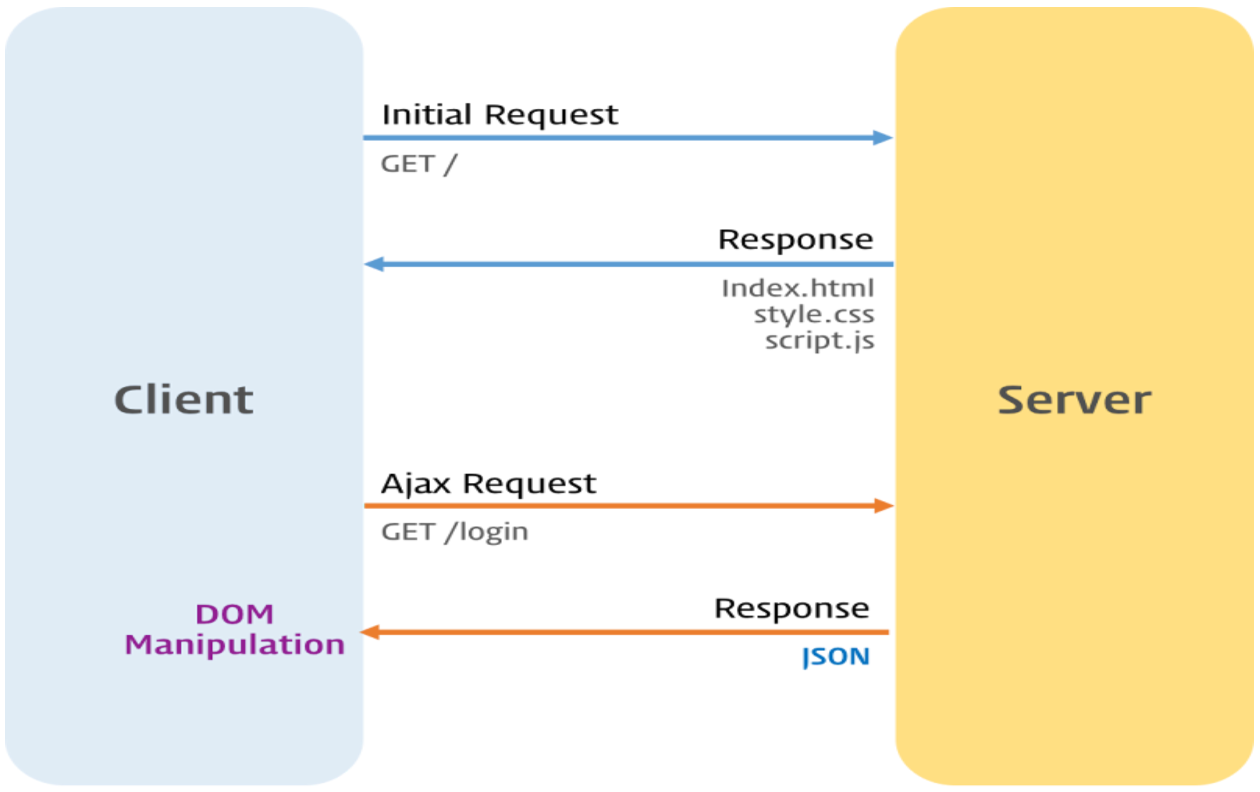
- A way to update web pages without reloading.

Why use AJAX?

- Makes web apps faster and more interactive.

Main Tools Used:

- XMLHttpRequest (old method)
- fetch API (modern method)



Difference Between Normal Request & AJAX Request

Normal Request (Without AJAX)

User Action → Click a button or submit a form .
Request Sent → The browser sends a request to the server .
Server Response → Sends a **new HTML page** (includes HTML, CSS, and JS) .
Page Reloads → The entire page refreshes and loads again.

📌 **Example:** When you submit a form, the server sends a completely new webpage, reloading everything.

AJAX Request (With AJAX)

User Action → Click a button or submit a form.
Request Sent → JavaScript (fetch()) sends the request **without reloading**.
Server Response → Sends only **data** (usually JSON, not a full webpage).
JavaScript Updates the Page Using DOM → The received JSON data is used to update specific parts of the webpage dynamically using **DOM manipulation**.

fetch(url)

fetch() is a function in JavaScript that helps you get data from the internet (APIs) without refreshing the page.
Fetch will accept two arguments

1. Url
2. Optional object

Why use fetch()?

- You can get data from a server
- You can send data to a server (e.g., saving a new user).
- It works in the background, so your website doesn't freeze.

Example →

```
fetch("https://jsonplaceholder.typicode.com/posts/1") // 1. Ask for data
 .then(response => response.json()) // 2. Convert response to JSON
 .then(data => console.log(data)) // 3. Use the data
 .catch(error => console.error("Error:", error)); // 4. Handle errors
```

Using Async Await

```
async function fetchData() {
 try {
   let response = await fetch("https://jsonplaceholder.typicode.com/posts/1"); // Send
request
   let data = await response.json(); // Convert response to JSON
   console.log(data); // Display data
 } catch (error) {
   console.error("Error:", error); // Handle errors
 }
}
fetchData();
```

ES6 Features

- Arrow Functions

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // Output: 8
```

- Template Literals

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

- Enhanced Object Literals

```
const name = "Alice";
const age = 25;
const person = { name, age }; // No need for name: name, age: age
console.log(person); // Output: { name: 'Alice', age: 25 }
```

- for of Loop

```
const numbers = [10, 20, 30];
for (const num of numbers) {
  console.log(num); // Output: 10, 20, 30
}
```

- Default Parameters

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

- let and const
- Modules (import/export)
- Symbol (New Primitive Type)
- Destructuring
- Spread and Rest Operators
- Classes
- Promises

Destructuring

Destructuring is a process of extracting data from an array or object and storing it into variables.

1. It Reduces code complexity
2. It Increases readability
3. For arrays: We can use any variable names.
4. For objects: We must use the exact key name (case-sensitive).

Example1 → nested array

```
const numbers = [10, [20, [30, 40], 50], 60];
// Destructuring multi-level nested arrays
const [a, [b, [c, d], e], f] = numbers;
console.log(a); // Output: 10
console.log(b); // Output: 20
console.log(c); // Output: 30
console.log(d); // Output: 40
console.log(e); // Output: 50
console.log(f); // Output: 60
```

Example2 → nested objects

```
const company = {
  name: "TechCorp",
  location: {
    country: "India",
    state: "Delhi",
    city: {
      name: "Delhi",
      zip: 94105,
    },
  },
  employees: {
    manager: {
      name: "Akshay",
      department: "IT",
    },
  },
};

// Destructuring multi-level nested objects
const {
  name: companyName,
  location: {
    country,
    state,
    city: { name: cityName, zip },
  },
  employees: {
    manager: { name: managerName, department },
  },
} = company;

console.log(companyName); // Output: TechCorp
console.log(country); // Output: India
console.log(state); // Output: Delhi
console.log(cityName); // Output: Delhi
console.log(zip); // Output: 94105
console.log(managerName); // Output: Akshay
console.log(department); // Output: IT
```

Spread

The spread syntax is used to expand(unpack) an iterable (like an array, string, or object) into its individual elements.

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray]; // creates a shallow copy

const arr1 = [1, 2];
const arr2 = [3, 4];
const combinedArray = [...arr1, ...arr2]; // [1, 2, 3, 4]

const myArray = [1, 2];
const newArray = [...myArray, 3, 4]; // [1, 2, 3, 4]

const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const mergedObject = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 } (properties from obj2 override obj1)
```

Rest

The rest parameters are used in function definitions mostly to collect multiple elements into a single array. They "gather" remaining arguments or properties.

```
function greet(greeting, ...names) {
   // names will be an array containing all arguments after 'greeting'
    return `${greeting}, ${names.join(' and ')}!`;
  }
console.log(greet("Hello", "Alice", "Bob", "Charlie")); // "Hello, Alice and Bob and Charlie!"
```

React

React JS

Modules

- Modules are used to break the large amount of code into multiple pieces.
- It is Easy to access, Modify and maintain the code.
- Whenever we use import and export keywords in a JavaScript file, It will become a
- module. So we have to specify type="module" in the script tag.
- Modules can be exported in 2 ways.
- They are
  - Default Export
  - Named Export

1. Default Export:

- We can export only one variable at a time.
- There should be only one default export in a file.
- While importing you can use any identifier name.

2. Named Export: