JavaScript

Index

Introduction of JavaScript

➔ What is JavaScript

- JavaScript is a Single threaded, Synchronous language client side scripting language.
- JavaScript is a lightweight programming language that web developers commonly use to create more dynamic interactions when developing web pages, applications, servers, and or even games.
- JavaScript is a high-level programming language that is commonly used in web development to create interactive web pages and dynamic user interfaces.
- JavaScript is a client-side language, meaning it runs in a user's web browser rather than on a web server. It can be used to add functionality to web pages, such as form validation, image sliders, and interactive menus. It can also be used to create more complex applications, such as web-based games and social media platforms.

➔ Feature of JavaScript

1. **Scripting Language -** Javascript executes the client-side script in the browser.
2. **Interpreter -** The browser interprets JavaScript code.
3. **Event Handling -** Events are actions. Javascript provides event-handling options.
4. **Light Weight -** As Javascript is not a compiled language, source code never changes to byte code before running time. Low-end devices can also run Javascript because of its lightweight feature.
5. **Case Sensitive -** In Javascript, names, variables, keywords, and functions are case-sensitive.
6. **Control Statements -** Javascript has control statements like if-else-if, switch case, and loop. Users can write complex code using these control statements.
7. **Objects as first-class Citizens -** Javascript arrays, functions, and symbols are objects which can inherit the Object prototype properties. Objects being first-class citizens means Objects can do all tasks.
8. **Dynamic Typing -** Javascript variables can have any value type. The same variable can have a string value, an integer value, or any other.
9. **Supports Functional Programming -** Javascript functions can be an argument to another function, can call by reference, and can assign to a variable.
10. **Client-side Validations -** Javascript client-side validations allow users to submit valid data to the server during a form submission.
11. **Platform Independent -** Javascript will run in the same way in all systems with any operating system.
12. **Async Processing -** Javascript async-await and promise features provide asynchronous nature. As the processes run in parallel, it improves processing time and responsiveness.

➔ Advantage and Disadvantage of JavaScript

Advantage	Disadvantage
Speed - JavaScript is an **“interpreted”** language, it reduces the time required by other programming languages like **(JAVA)** for compilation. JS is also a client-side script, speeding up the execution of the program as it saves the time required to connect to the server.	**Client-side Security -** Since the JavaScript code is viewable to the user, others may use it for malicious purposes. These practices may include using the source code without authentication. Also, it is very easy to place some code into the site that compromises the security of data over the website.

Simplicity - JavaScript is easy to understand and learn. The Structure is simple for the users as well as the developers. It is also very feasible to implement, saving developers a lot of money for developing dynamic content for the web.	**Browser Support -** The browser interprets JavaScript differently in different browsers. Thus, the code must be run on various platforms before publishing. The older browsers don't support some new functions and we need to check them as well.
Popularity - Since all modern browsers support JavaScript, it is seen almost everywhere. All the famous companies use JavaScript as a tool including ***Google, Amazon, PayPal etc.***	**Cannot Debug -** Although some HTML editors allow for debugging, they are not as effective as editors for C or C++. Additionally, the developer has a difficult time figuring out the issue because the browser doesn't display any errors.
Server Load - As JavaScript operates on the client-side, data validation is possible on the browser itself rather than sending it off to the server. In case of any discrepancy, the whole website needs not to be reloaded. The browser updates only the selected segment of the page.	**Inheritance -** JavaScript does not support multiple inheritance; only one inheritance is supported. This property of object-oriented languages might be necessary for some programmes.
Less Overhead - JavaScript improves the performance of websites and web applications by reducing the code length. The codes contain less overheads with the use of various built-in functions for loops, DOM access, etc.	**Rendering Stopped -** A single code error can stop the rendering of the entire JavaScript code on the website. To the user, it looks as if JavaScript was not present. However, the browsers are extremely tolerant of these errors.

➔ Var, Let and Const

- In JavaScript, **Var, Let, and Const** are keywords used to declare variables. Each keyword has its own behavior and scoping rules.

- **Var:**
  - **Var** is a keyword used to declare variables in JavaScript.
  - **Var** is **function-scoped**, meaning that a variable declared with var inside a function is accessible throughout that entire function.
  - If **Var** is declared outside of any function, it becomes globally scoped.
  - If you declare a variable with **Var** twice in the same function, the second declaration will overwrite the first.

```
// Reinitialized
var x = 10;
var x = 15;
function example() {
    var x = 20;
    console.log(x); // output: 20
}
example();
console.log(x); // output: 15
```

```
// Redeclaring
var x = 10;
x = 15;
function example() {
     var x = 20;
    console.log(x); // output: 20
}
example();
console.log(x); // output: 15
```

```
// Block Scope
{
    var a = 20;
    console.log(a);   // 20
}
console.log(a);     // 20
```

- **Let:**
    - **Let** is a keyword used to declare variables in JavaScript, just like **Var**.
    - **Let** is block-scoped, meaning that a variable declared with let inside a block (e.g. a loop, an if statement, or a function) is only accessible within that block.
    - If you declare a variable with **Let** twice in the same block, an error will be thrown.

```
// Reinitialized
let count = 10;
let count = 15;
function example() {
    let count = 20;
    console.log(count);
    // output: 20
}
example();
console.log(count);
// output: SyntaxError:
Identifier 'count' has
already been declared
```

```
// Redeclaring
let count = 10;
count = 15;
function example() {
    let count = 20;
    console.log(count);
    // output: 20
}
example();
console.log(count);
// output: 15
```

```
// Block Scope
{
    let a = 20;
    console.log(a);
    // Output: 20
}
console.log(a);
// Output: ReferenceError: a
is not defined
```

- **Const:**
    - **Const** is a keyword used to declare constants in JavaScript. A constant is a variable that cannot be reassigned once it has been initialized.
    - Like **Let, Const** is also block-scoped.
    - When you declare a constant, you must initialize it with a value.

```
// Reinitialized
const count = 10;
const count = 15;
function example(){
    const count = 20;
    console.log(count);
    // output: 20
}
example();
console.log(count);
// output: SyntaxError:
Identifier 'count' has
already been declared
```

```
// Redeclaring
const count = 10;
count = 15;
function example(){
    const count = 20;
    console.log(count);
    // output: 20
}
example();
console.log(count);
// output: TypeError:
Assignment to constant
variable.
```

```
// Block Scope
{
    const a = 20;
    console.log(a);
    // Output: 20
}
console.log(a);
// Output: ReferenceError: a
is not defined
```

➔ Difference Between Var, Let and Const

	Var	Let	Const
Introduce	Pre ES2015	ES2015(ES6)	ES2016(ES6)
Scope	Globally scope and function scoped	Globally scope and block scope	Globally scope and block scope
Hoisting	**Var** is hoisted to top of its execution (either global or function) and initialized as undefined.	Let is hoisted to top of its execution (either global or block) and left uninitialized.	**Const** is hoisted to the top of its execution (either global or block) and left uninitialized.
Redeclaration within scope	**YES**	**NO**	**NO**
Reassigned within scope	**YES**	**YES**	**NO**

➔ JavaScript Variable Scope -

- The scope of a variable is the region of your program in which it is defined.
- **Three types of scope in JavaScript.**
  - **Global Scope:** The global scope is accessible from any part of your code, including functions and other scopes. Any variable or function declared outside of a function or block of code is in the global scope. Global variables are accessible from any part of your code, but they can be modified by any part of your code as well, which can lead to unexpected behavior.
  - **Local Scope:** Local scope refers to variables or functions that are declared within a function or block of code. These variables and functions are only accessible within that function or block of code. Local scope is important because it prevents naming conflicts between different parts of your code.
  - **Block Scope:** Block scope refers to variables or functions that are declared within a block of code, such as an if statement or a for loop. Block scope was introduced in ES6 with the let and const keywords. Variables declared with let or const are only accessible within the block of code they are declared in. This helps to prevent naming conflicts and can make your code easier to understand and maintain.

```
// Global Scope
let gScope = "Hello World";
function GlobalFunc(){
console.log(gScope);
    // Output: Hello World
```

```
// Global Scope
let globalScope = "Hello World";
function LocalFunc(){
    // Local Scope
    var lScope = "PrepBytes"
    console.log(lScope);// Output: PrepBytes
}
LocalFunc();
```

```
// Block Scope
for(let i = 0 ; i < 10 ; i++){
    // Here i Follow Block Scope
    console.log(i)
    // Output: 1 to 9
}
console.log(i);
// Output: ReferenceError: i is
```

}
GlobalFunc();
		not defined

➔ **Operator and Type of Operator in JavaScript**

- In JavaScript, an operator is a special symbol used to perform operations on operands (values and variables).

- **There are seven types of operators**

  - **Assignment Operators** - These are used to assign values to variables. The most common assignment operator is '='.

```
// Assign a value to a variable
let value = 10;
// Assign a variable to a variable
let variable = "Shubham"
let getValue = variable;
```

  - **Arithmetic Operators** - These are used to perform mathematical operations like addition, subtraction, multiplication, and division.

// Addition
let a = 5;
let b = 10;
console.log(a + b);
// Output: 15
	// Subtraction
	let a = 10;
	let b = 3;
	console.log(a - b);
	// Output: 7

// Multiplication
let a = 10;
let b = 3;
console.log(a * b);
// Output: 30
	// Division
	let a = 10;
	let b = 3;
	console.log(a / b);
	// Output: 3.33

// Modulus
let a = 10;
let b = 3;
console.log(a % b);
// Output: 1	

  - **Comparison Operators** - These are used to compare two values and return a Boolean value (true or false).

// Equal to (==): checks if the two operands are equal.	// Not equal to (!=): checks if the two operands are not equal.

```
let a = 10;
let b = 10;
console.log(a == b);
// Output: True
```

```
let a = 10;
let b = 10;
console.log(a != b);
// Output: False
```

```
// Greater than (>): checks if the
first operand is greater than the
second operand.
let a = 10;
let b = 5;
console.log(a > b);
// Output: true
```

```
// Less than (<): checks if the first
operand is less than the second
operand.
let a = 10;
let b = 5;
console.log(a < b);
// Output: false
```

```
// Greater than or equal to (>=):
checks if the first operand is
greater than or equal to the second
operand.
let a = 10;
let b = 5;
console.log(a >= b);
// Output: true
```

```
// Less than or equal to (<=): checks
if the first operand is less than or
equal to the second operand.
let a = 10;
let b = 5;
console.log(a <= b);
// Output: false
```

○ **Logical Operators** - These are used to perform logical operations such as AND (&&), OR (||), and NOT (!).

```
// AND (&&): returns
true if both operands
are true.
let a = 5;
let b = 10;
console.log(a < b && a
!= b);
// Output: true
```

```
// OR (||): returns true
if at least one operand
is true.
let a = 5;
let b = 10;
console.log(a > b || a
!= b);
// Output: true
```

```
// NOT (!): returns the
opposite of the Boolean
value of the operand.
let a = 5;
let b = 10;
console.log(!(a < b));
// Output: false
```

○ **Bitwise Operators** - These are used to perform bitwise operations on binary numbers.

```
// Bitwise AND (&): performs a
bitwise AND operation on two
operands.
let a = 5;
let b = 3;
console.log(a & b);
// Output: 1
```

```
// Bitwise OR (|): performs a bitwise
OR operation on two operands.
let a = 5;
let b = 3;
console.log(a | b);
// Output: 7
```

- **Conditional (ternary) operator** - This operator allows for a concise way of writing an if...else statement.

```
// Syntax - condition ? value1 : value2
```

```
let age = 20;
let message = age >= 18 ? "You are an adult" : "You are a minor";
console.log(message);
// Output: "You are an adult"
```

```
let score = 80;
let grade = score >= 90 ? "A" : score >= 80 ? "B" : score >= 70 ? "C" : "F";
console.log(grade);
// Output: "B"
```

.

- **Type Operator** - Type Operators: These are used to determine the type of a value. The most common type operator is typeof.

```
// Syntax - typeof operand
```

```
let name = "John";
console.log(typeof name);
// Output: "string"
```

```
let id = 10023555;
console.log(typeof id);
// Output: "number"
```

➔ **Conditional Statements** - Conditional statements are used in programming to execute certain code blocks only if a specified condition is met. In JavaScript, there are three main types of conditional statements: **if** statements, **if...else** statements, and **switch** statements.

- **if statement** - The **if** statement checks if a condition is true, and if it is, the code block inside the statement is executed. If the condition is false, the code block is skipped.

```
let variable = 1;
let value = true;
if(value == variable){
    console.log("if statement called");
}
```

- **if...else statement** - The if...else statement checks if a condition is true. If it is, the code block inside the if statement is executed. If the condition is false, the code block inside the else statement is executed.

```
let variable = 1;
let value = true;
if(value === variable){
    console.log("if statement called...");
}
else{
    console.log('else statement called...');
}
```

- **switch statement** - The switch statement is used to compare a value against multiple possible values and execute different code blocks depending on the match.

```
// Syntax of Switch case
switch (expression) {
    case value1:
      // code to be executed if expression matches value1
      break;
    case value2:
      // code to be executed if expression matches value2
      break;
    // ...
    default:
      // code to be executed if expression doesn't match any value
  }
```

```
let day = "Tuesday";
switch (day) {
  case "Monday":
    console.log("Today is Monday");
    break;
  case "Tuesday":
    console.log("Today is Tuesday");
    break;
  case "Wednesday":
    console.log("Today is Wednesday");
    break;
  case "Thursday":
    console.log("Today is Thursday");
    break;
  case "Friday":
    console.log("Today is Friday");
    break;
  case "Saturday":
  case "Sunday":
    console.log("It's the weekend!");
    break;
  default:
    console.log("Invalid day");
}
```

➔ Data Type in JavaScript

- **Two Type of Data Type in JavaScript**
  - **Primitive Data Type**
    - **Number** - A numerical value, including integers and floating-point numbers.
    - **String** - A sequence of characters enclosed in single or double quotes.
    - **Boolean** - A true/false value.
    - **Null** - A special value that represents no value or empty value.
    - **Undefine** - A variable that has been declared but not assigned a value.
    - **Symbol** - A unique value that is used as an identifier for object properties.

```
let myName = 'John';
let myAge = 30;
let isStudent = true;
let myVar = null;
let mySymbol = Symbol('description');

console.log(typeof myName); // Output: string
console.log(typeof myAge); // Output: number
console.log(typeof isStudent); // Output: boolean
console.log(typeof myVar); // Output: object
console.log(typeof mySymbol); // Output: symbol
```

  - **Non-Primitive Data Type (Array, Object and it's Method)**
    - **Object**: A javaScript object is an entity having state and behavior (properties and method).
      - JavaScript Object is a collection of key and value pairs.
      - JavaScript is an Object based programming language, Everything is an Object in JavaScript.

    - There are three ways to create an Object in JavaScript.
      - **By Object literal**
      - **By creating instance** of Object directly (using new keyword)
      - **By using an object constructor** (using a new keyword) - you need to create a function with arguments. Each argument value can be assigned in the current object by using this keyword.

```
// By Object
literal
```

```
const obj = {
    Name: 'Shubham',
    Age: 12,
    Year: 2023,
```

	`    Dept: 'Elevation'`
	`}`

`// By creating instance of Object directly (using new keyword)`	`const Student = new Object();`
	`Student.name = 'Shubham';`
	`Student.Year = 2021`

`// By using an object constructor`	`function Emp(id, name, year){`
	`    this.id = id,`
	`    this.name = name,`
	`    this.year = year`
	`}`
	`Const employee = new Emp(1, 'Shubham', 2023);`
	`By using  (employee.id, employee.name and employee.year)`
	`we can access the property and value of Object.`

- **Method of Object**

S.No	Object Methods	Description
1.	Object.assign()	This method is used to copy enumerable and own properties from a source object to a target object
2.	Object.create()	This method is used to create a new object with the specified prototype object and properties.
3.	Object.entries()	This method returns an array with arrays of the key, value pairs.
4.	Object.is()	This method determines whether two values are the same value.
5.	Object.keys()	This method returns an array of a given object's own property names.
6.	Object.values()	This method returns an array of values.

- **Array:**
  - JavaScript array is an object that represents a collection of similar types of elements.
  - Arrays are used to store multiple values in a single varianle. Each item in an array has a number attached to it, called a numeric index, that allows us to excess the elements.

  - **There are three ways to create Array** -
    - **By Array literal**
    - By creating instance of Array directly (using **New keyword**)
    - By using an **Array Constructor** (using new keyword).
      - By using indices we are able to access the index and index value.

`// By array literal`	`const array = [10, 20, 30, 40]`
	`console.log(array);`

`// By creating instance of Array directly (using new keyword)`	`const ArrName = new Array();`
	`ArrName[0] - 'Shubham';`
	`ArrName[1] - 'Kumar'`

`// By using an Array Constructor (using new keyword).`	`const Emp = new Array('Shubham', 'Kumar', 'Singh');`
	`console.log(Emp);`

- **Array Method**

S. No	Method	Description
1.	**concat()**	It returns a new array object that contains two or more merged arrays.
2.	**every()**	It determines whether all the elements of an array are satisfying the provided function conditions.
3.	**fill()**	It fills elements into an array with static values.
4.	**filter()**	It returns the new array containing the elements that pass the provided function conditions.
5.	**find()**	It returns the value of the first element in the given array that satisfies the specified condition.
6.	**forEach()**	It invokes the provided function once for each element of an array.
7.	**join()**	It joins the elements of an array as a string.
8.	**map()**	It calls the specified function for every array element and returns the new array
9.	**reduce()**	It executes a provided function for each value from left to right and reduces the array to a single value.
10.	**push()**	It adds one or more elements to the end of an array.
11.	**pop()**	It removes and returns the last element of an array.
12.	**reverse()**	It reverses the elements of a given array.
13.	**shift()**	It removes and returns the first element of an array.
14.	**unshift()**	It adds one or more elements in the beginning of the given array.
15.	**slice()**	It returns a new array containing the copy of the part of the given array.

➔ **Difference Between Map, Filter and Reduce**

```
// Example of Map, Filter and Reduce
let array = [1,2,3,4,5,6,7,8,9, 10]
```

```
// Using Map Method of Array
array.map((item)=> {
    console.log(item);
    // Output - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
})
```

```
// Using filter Method
let lessFive = array.filter((data) => {
    return data <= 5
})
console.log(lessFive);
// Output - [ 1, 2, 3, 4, 5 ]
```

```
// Reduce
let sumrOfArray = array.reduce((prevValue, currentValue) => {
    return prevValue + currentValue
},0)
console.log(sumrOfArray);
// Output - 55
```

○ String

- **String:** The JavaScript string is an object that represents a sequence of characters.
- In JavaScript, a string is a sequence of characters enclosed in single quotes, double quotes, or backticks. String in JavaScript are immutable, Which means that once a string is created, it cannot be changed.
- There are **Two ways** to create String in JavaScript
    - **By String Literal -** The string literal is created with double quotes or single quotes.
    - The syntax of creating a string object using a new keyword. **New Keyword** is used to create an instance of String.

`// By String Literal`	`let Name = "PrepBytes";`
	`console.log(Name);`

`// By String Object (using New Keyword)`	`let strName = new String('Namaste JavaScript')`
	`console.log(strName);`

- **String Method:**

S No.	String Method	Description
1.	**toLowerCase()**	It converts the given string into lowercase letters.
2.	**toUpperCase()**	It converts the given string into uppercase letters.
3.	**split()**	It splits a string into a substring array, then returns that newly created array.
4.	**toString()**	It provides a string representing the particular object.
5.	**slice()**	It is used to fetch the part of the given string. It allows us to assign positive as well as negative indexes.
6.	**substring()**	It is used to fetch the part of the given string on the basis of the specified index.
7.	**replace()**	It replaces a given string with the specified replacement.
8.	**match()**	It searches a specified regular expression in a given string and returns that regular expression if a match occurs.
9.	**trim()**	It trims the white space from the left and right side of the string.
10.	**concat()**	It provides a combination of two or more strings.
11.	**charAt()**	It provides the char value present at the specified index.
12.	**charCodeAt()**	It provides the Unicode value of a character present at the specified index.

➔ **Syntax and Example of String Method**

`toLowerCase()`	`let myName = "PrepBytes"`
	`console.log(myName.toLowerCase());`
	`// Output - prepbytes`

`toUpperCase()`	`let str = "prepbytes";`
	`console.log(str.toUpperCase());`
	`// Output - PREPBYTES`

`Split()`	`let myName = "Shubham Kumar Singh";`
	`let str = myName.split(' ');`
	`console.log(str);`
	`// Output - 'Shubham', 'Kumar', 'Singh'`

`toString()`	`let myName = "Shubham Kumar"`
	`let num = 20;`
	`console.log(typeof num.toString());`

	`// Output - String`
	`console.log(myName.toString());`
	`// Output - Shubham Kumar`

`Slice()`	`let course = 'JavaScript';`
	`console.log(course.slice(0, 4));`
	`// Output - Java`

`substring()`	`let Course = "JavaScript";`
	`console.log(Course.substring(4,10));`
	`// Output - Script`

`replace()`	`let myName = 'Shubham Kumar'`
	`console.log(myName.replace('Kumar', 'Singh'));`
	`// Output - Shubham Singh`

`Match()`	`let greet = 'Hi I am Shubham Kumar.'`
	`console.log(greet.match('Kumar'));`
	`// Kumar : Index - 16 First Match and return word and index`
	`console.log(greet.match(/Hi/));`
	`// Second syntax to write`
	`console.log(greet.match(/Hi/g));`
	`// g represent global`

`trim()`	`let str = '               Shubham Kumar              ';`
	`console.log(str.trim());`

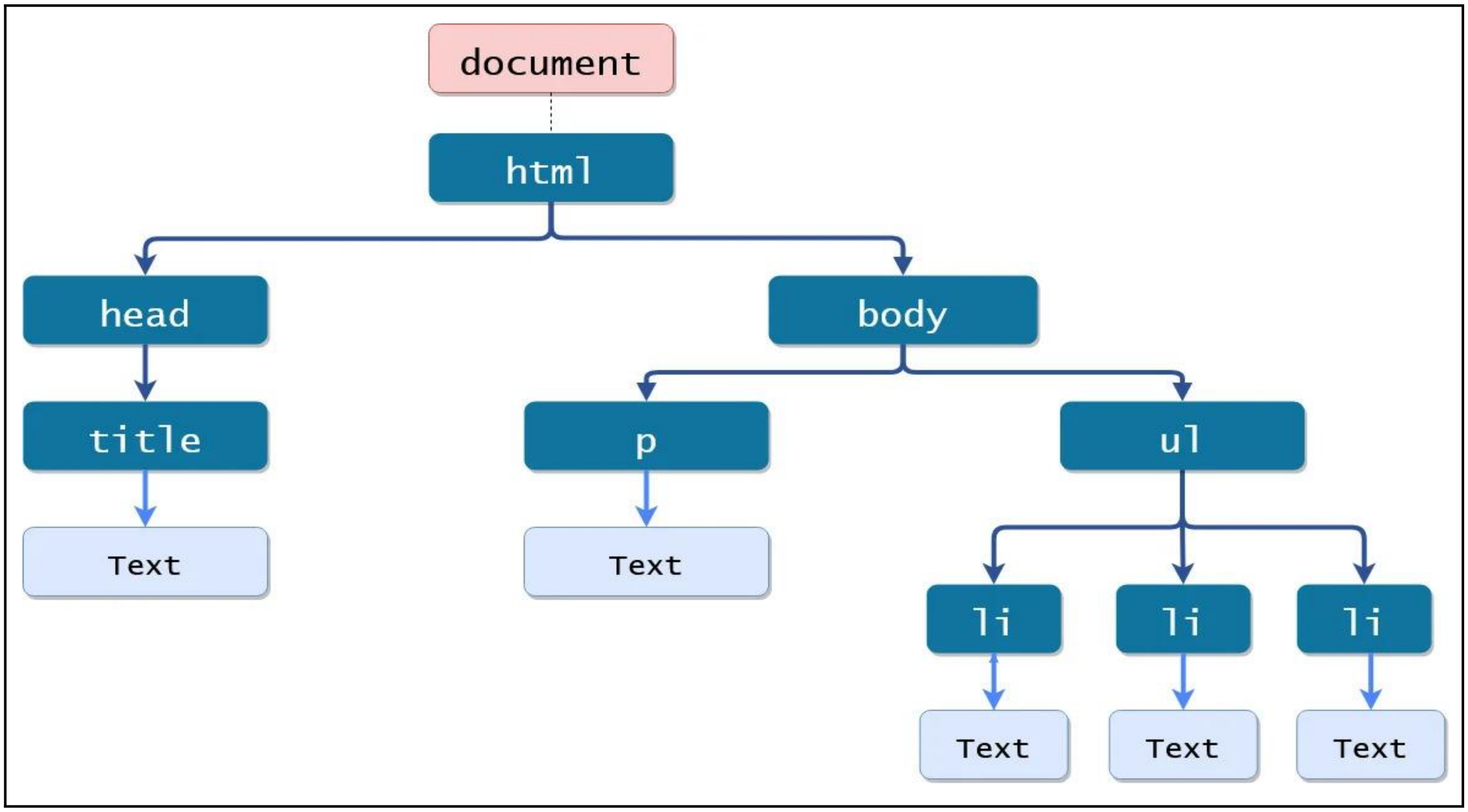
`// concat()`	`let str1 = "Prep",`
	`let str2 = 'Bytes'`
	`console.log(str1.concat(str2)); // Output - Shubham Kumar`

`// charAt()`	`let str = "PrepBytes";`
	`console.log(str.charAt(4)); // Output - B`

`// charCodeAt()`	`let str = "PrepBytes";`
	`console.log(str.charCodeAt(0)); // Output - ASCII of P - 80`
	`console.log(str.charCodeAt(4)); // Output - ASCII of B - 66`

➔ DOM - (Document Object Model)

- **DOM -** The **Document Object Model (DOM)** is a programming interface for web documents. It represents the page so that programs can **change** the **document structure**, **style**, and **content**. The DOM represents the document as nodes and objects, and that way, programming languages can interact with the page.
  - The Document Object represents the whole html document.
  - When an html document is loaded in the browser, it becomes a document object. It is the root element that represents the html document. It has properties and methods. By the help of document objects, we can add dynamic content to our web page.
  - `Window.document` is the same as the `document`.



❖ **DOM (Document Object Model) has some methods.**

➢ **getElementById() - getElementById** is a method in the JavaScript Document Object Model (DOM) that is used to select and return a reference to a single element in the HTML document that matches a specific ID attribute.

```
document.getElementById("html id");
```

```
let idValue = document.getElementById('Pass id here');
```

➢ **getElementsByClassName() - getElementsByClassName** is a method in the JavaScript Document Object Model (DOM) that is used to select and return a collection of all the elements in an HTML document that match a specific class name.

```
document.getElementsByClassName('HTML Class Name');
```

```
const classValue = document.getElementsByClassName('Pass Class Name Here');
```

➢ **querySelector()** - querySelector is a method in the JavaScript Document Object Model (DOM) that allows you to select a single element in an HTML document using a CSS selector. It returns the first element that matches the specified selector.

```
document.querySelector('HTML Tag name');
```

```
document.querySelector('HTML ID name with Symbol #')
```

```
document.querySelector('HTML class name with Symbol .')
```

```
let idVal = document.querySelector('#item_demo')
```

➢ **querySelectorAll()** - querySelectorAll is a method in the JavaScript Document Object Model (DOM) that allows you to select one or more elements in an HTML document using a CSS selector. It returns a NodeList object containing all the elements that match the specified selector.

```
document.querySelectorAll('HTML Tag name');
```

```
document.querySelectorAll('HTML class name with Symbol .')
```

```
let classVal = document.querySelector('.item_demo')
```

```
let tagVal = document.querySelector('h1')
```

➢ **getElementsByTagName()** – The **getElementsByTagName** method returns a collection of all elements with a specified tag name. It returns an HTMLcollection.

```
document.getElementsByTagName('Pass the HTML tag')
```

```
let h1Val = document.getElementsByTagName('h1')
```

➢ **createElement():** As the name goes, it is used to create an element & place it anywhere in the DOM structure.

```
// Dynamically Create a list
var myNewListItem = document.createElement('li')
```

```
// Dynamically Create a div
var myNewProd = document.createElement('div');
```

➢ **appendChild():** Previously we created an element, now we will add two elements to our list of links using appendChild.

```
// List assign to a variable
var myNewListItem = document.createElement('li')
myLinkList.appendChild(myNewListItem);
```

```
// List assign to Body tag
var myNewProd = document.createElement('div');
body.appendChild(myNewProd)
```

➢ **innerHTML:**The innerHTML property can be used to write the dynamic html on the html document. It is mostly used to add or append any variable or HTML elements.
   ■ We can assign a html tag to any element.

```
// Assign variable to html elements
let val = "PrepBytes"
let idVal = document.getElementById('get');
idVal.innerHTML = val;
```

```
// Assign html element to html elements
let idVal = document.getElementById('get');
idVal.innerHTML = `<h1>PrepBytes</h1>`;
```

➢ **innerText:** The innerText property can be used to write the dynamic html on the html document. This is similar to innerHTML.
   ■ We can assign some html tags dynamically by using innerText.

```
// assign some text to html tag by using innerText
let idVal = document.getElementById('get');
idVal.innerText = "PrepBytes CollegeDekho";
```

```
// innerText return content of html tag
let idVal = document.getElementById('get');
let content = idVal.innerText;
console.log(content);
```

➢ **setAttributes:** A useful method to replace values in the attribute. Assigning a new value to an existing attribute is done using setAttribute. Suppose we have an attribute “abc” containing the value “Best.”

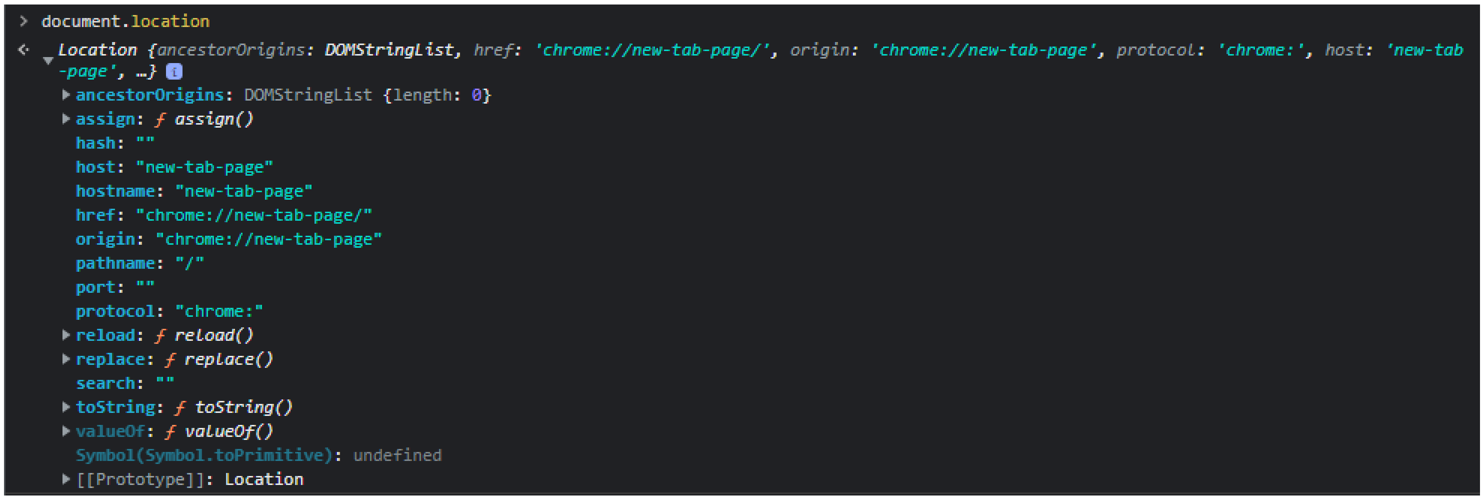
```
var myLinkFive = document.getElementById("HTML ID");
myLinkFive.setAttribute('abc', 'Awesome');
```

➢ **getAttributes –** With the getAttribute method, you can access the value of any attribute of an element on a page. Suppose there's a div with an id attribute having value “Best.”

```
var myLinkFive = document.getElementById("get");
let getAttributeValue = myLinkFive.getAttribute("class");
console.log(getAttributeValue);
// Output : html class name best
```

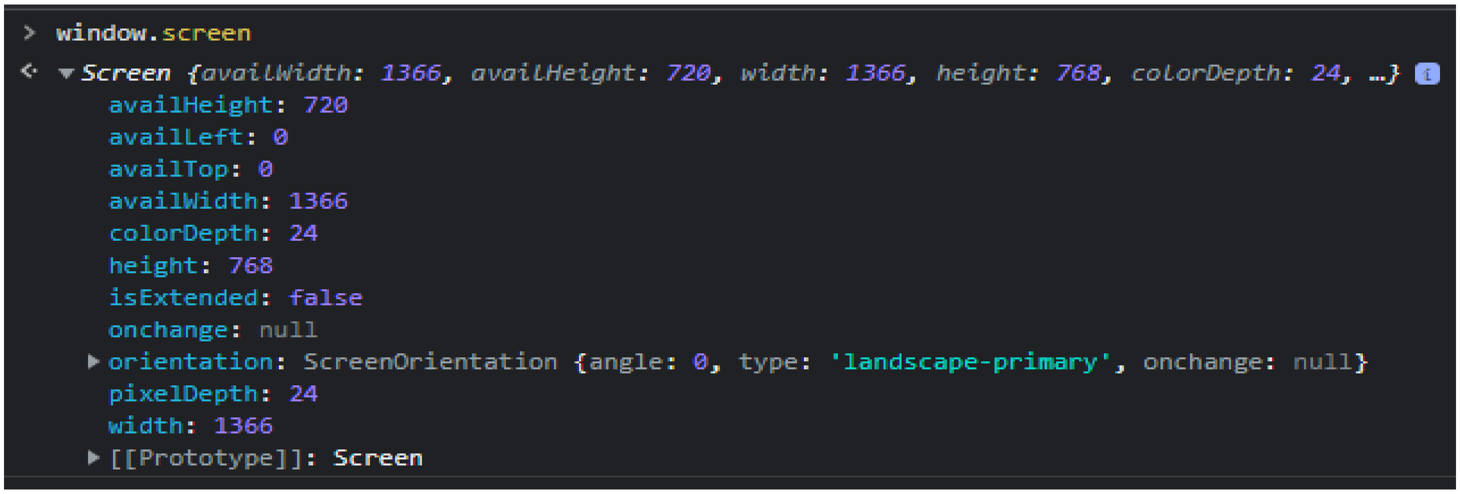
➔ Window vs Document

- **Document -** A document is an object inside the window object and we use a document object for manipulation inside the document.



```
> document.location
<· ▾Location {ancestorOrigins: DOMStringList, href: 'chrome://new-tab-page/', origin: 'chrome://new-tab-page', protocol: 'chrome:', host: 'new-tab
   -page', …}
   ▸ancestorOrigins: DOMStringList {length: 0}
   ▸assign: ƒ assign()
    hash: ""
    host: "new-tab-page"
    hostname: "new-tab-page"
    href: "chrome://new-tab-page/"
    origin: "chrome://new-tab-page"
    pathname: "/"
    port: ""
    protocol: "chrome:"
   ▸reload: ƒ reload()
   ▸replace: ƒ replace()
    search: ""
   ▸toString: ƒ toString()
   ▸valueOf: ƒ valueOf()
    Symbol(Symbol.toPrimitive): undefined
   ▸[[Prototype]]: Location
```

- **Window** - The window object represents a window in the browser. An object of the window is created automatically by the browser. Windows is the object of the browser, it is not the object of javascript. The javascript objects are string, array, date etc.

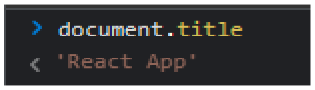
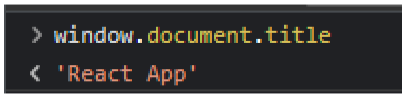


```
> window.screen
<· ▾Screen {availWidth: 1366, availHeight: 720, width: 1366, height: 768, colorDepth: 24, …}
    availHeight: 720
    availLeft: 0
    availTop: 0
    availWidth: 1366
    colorDepth: 24
    height: 768
    isExtended: false
    onchange: null
   ▸orientation: ScreenOrientation {angle: 0, type: 'landscape-primary', onchange: null}
    pixelDepth: 24
    width: 1366
   ▸[[Prototype]]: Screen
```

➔ **The Difference between Window and Document -**

Document	**Window**
It represents the document loaded inside the window or browser.	It represents the browser window in which you are seeing the content.
The properties related to it are stored in the document object.	The properties related to it are stored in the window object.
It is loaded after the loading window because the window contains a document.	It is loaded before the document because of the window container document.
It is the root element of the document object model.	The window is the global element for all objects, functions, etc.
It is an object of the window.	It is an object of the browser.
We can not access windows objects properties inside the document.	We can access document object properties inside the window.
Example: document.title will return the title of the document	Example: window.document.title will return the title of the document.

logically:
 	
document:{ properties}
	logically:
	window:{
	document:{properties}
	}

 `> document.title` `< 'React App'`	 `> window.document.title` `< 'React App'`

➔ JavaScript Loop and Functions

◆ JavaScript Loops-

- **Loop** - Javascript loop, is used to iterate the piece of code using for, while, do While, or for-in loop. It makes the code compact. It is mostly used in JavaScript Arrays.

➔ **There are five types of loops in JavaScript**

- **For loop** - The JavaScript for loop iterates the elements for the fixed number of times. It should be used if the number of iterations is known.

Syntax	`for(loop starting condition ; loop Condition ; increment/ decrement){`
	`    // Code We want to execute.`
	`}`

Example	`for(let i = 0 ; i < 5 ; i++){`
	`    // write the logic or`
	`    console.log(i);`
	`}`

- **While Loop -** The JavaScript **While Loop** iterates the elements for an infinite number of times. It should be used if the number of iterations is not known.

Syntax	`while(Condition){`
	    `//Code we want to execute.`
	`}`

Example	`while(n>0){`
	    `n = n % 10;`
	    `rem += n;`
	    `n = n / 10;`
	`}`

- **Do While loop -** The JavaScript do while loop iterates the elements for an infinite number of times like while loop. But, code is executed at least once whether the condition is true or false.

Syntax	`do{`
	    `// Code we want to execute`
	`} while(condition)`

Example	`do{`
	    `// Write code here`
	    `console.log('Hello World');`
	`}while(n=10)`

- **For in Loop -** The JavaScript for in loop is used to iterate the properties of an object.

Syntax	`for(let key of Object){`
	    `// write the code here`
	`}`

Example	`const obj = {name: "Shubham",roll: 100123 }`
	`for(let key in obj){`
	    ``console.log(`${key} : ${obj[key]}`);``
	`}`

- **For of loop -** The JavaScript for of loop is used to iterate the Object like (Array, Set, Map, String etc.)

Syntax	`for (element of iterable) {`
	    `// Write the logic.`
	`}`

Example	`const arr = [10, 20, 30, 40, 50];`

	`for(key of arr){`
	`    console.log(key);`
	`}`

➔ Function in JavaScript

- Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure, a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output.
- Functions are used to perform operations. We can call JavaScript functions many times to reuse the code.

Syntax	`Function nameOfFunction(parameter1, parameter2…..`
	`parameterN){`
	`    // write the code as you want.`
	`}`

Example1 This function console its value	`function Sum(num1, num2){`
	`    console.log(num1+num2);`
	`}`
	`Sum(10, 20); // invoked the function`

Example2 This function return a value	`function multiply(num1, num2){`
	`    let multi  = num1 * num2;`
	`    return multi;`
	`}`
	`let result = multiply(10, 20); // invoked the function`
	`console.log(result);`

➔ There are three ways to write function -

- **Function Declaration** - Function Declaration is the traditional way to define a function. It is somehow similar to the way we define a function in other programming languages. We start declaring using the keyword “function”. Then we write the function name and the parameters.

```
// Function Declaration
function Sum(a, b){
  console.log(a+b);
}
// Calling Function
Sum(5, 10)
```

- **Function Expression -** Function Expression is another way to define a function in JavaScript. Here we define a function using a variable and store the returned value in that variable.

```
// Function Expression
const Sum = function(a, b){
  console.log(a+b);
}
// Calling Function
Sum(5, 10)
```

- **Arrow Function -** Arrow functions have been introduced in the ES6 version of JavaScript. It is used to shorten the code. Here we do not use the “function” keyword and use the arrow symbol.

```
// Array Function
const Multi = (a, b) => a * b;
const result = Multi(10, 20);
console.log(result);
```

➔ There are different type of Function in JavaScript

- **Call Back Function-** A callback function is a function that is passed as an argument to another function and is called when that function completes its task. Callback functions are commonly used in asynchronous programming to handle the results of a task that may take some time to complete.

```
// Call Back Function
function calculate(n1, n2, callback){
  const result = n1 + n2;
  callback(result);
}
function printResult(output){
  console.log(`The result is: ${output}`);
}
calculate(10, 25, printResult)
```

- **Higher-order functions-** These are functions that take one or more functions as arguments or return a function as their result.

```
// Higher Order Function
function random(){
    return Math.floor(Math.random() * 100);
 }
  function player(name, id){
    return `${name} and ${id()}`
 }
  const p1 = player("Shubham", random);
```

```
console.log(p1);
// Output - Shubham and Random Number
```

- **Anonymous functions -** These are functions that do not have a name and are typically used as arguments for other functions or as immediately invoked function expressions (IIFEs).

```
// Anonymous functions
const Sum = function(a, b){
    return a / b
}
console.log(Sum(10, 5))
// Output - 2
```

- **Immediately Invoked Function Expressions (IIFEs):** These are functions that are executed as soon as they are defined.

```
(function(){
    console.log("This is Immediately Invoked Function Expressions");
})();
// Output - This is Immediately Invoked Function Expressions
```

- **First Class Function:** In JavaScript, functions are first-class citizens, which means that they can be treated like any other variable. This includes being passed as an argument to another function, returned as a value from a function, and assigned to a variable.

```
function sayHello(name) {
    console.log(`Hello, ${name}!`);
}
// assign the function to a variable
const greeting = sayHello;
// call the function through the variable
greeting("PrepBytes");
// outputs "Hello, PrepBytes!"
```

- **Generator Function:** In JavaScript, a generator function is a special type of function that allows you to control the execution of your code in a more granular way than with a regular function.

```
function * fun(){
    yield 10;
    yield 20;
    yield 30;
}
```

```
// Calling the Generate Function
var gen = fun();
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
```

→ Hoisting in Javascript:

→ **Hoisting –** Hoisting is a term used in JavaScript that describes how variable and function declarations are processed during the compilation phase. Specifically, it refers to the way that declarations are moved to the top of their respective scopes, allowing you to use them before they are actually declared in the code.

→ Here Some important points to understand about hoisting in JavaScript -

1. Declarations are processed before code execution: JavaScript engines process variable and function declarations before executing any code. This means that declarations are processed during the compilation phase, which occurs before the execution phase.
2. Variable declarations are hoisted but not their assignments: When you declare a variable using the var keyword, the variable declaration is hoisted to the top of its scope. However, the assignment of the variable is not hoisted. This means that you can use the variable before it's assigned a value, but you will get undefined if you try to access its value.
3. Function declarations are hoisted entirely: When you declare a function using the function keyword, the entire function declaration is hoisted to the top of its scope. This means that you can call the function before it's declared in the code.

```
console.log(myVar);
// Output: undefined
var myVar = "Hello, world!";
console.log(myVar);
// Output: "Hello, world!"
```

```
sayHello();
function sayHello() {
  console.log("Hello, world!");
}
// Output: "Hello, world!"
```

→ Closure In JavaScript

→ **Closure –** A closure can be defined as a JavaScript feature in which the inner function has access to the outer function variable. In JavaScript, every time a closure is created with the creation of a function.

→ **The closure has three scope chains listed as follows:**

- Access to its own scope.
- Access to the variables of the outer function.
- Access to the global variables.

```
function outerFunction() {
    var outerVariable = "Outer
Function Variable";
    function innerFunction() {
        console.log(outerVariable);
    }
    return innerFunction;
}
```

```
let globalVar = "Global Variable"
function outerFunction() {
    let outerVar = "Outer Function
Variable";
    function innerFunction() {
        let innerVar = "Inner
Function Variable"
        console.log(globalVar);
```

```
var innerFunc = outerFunction();
innerFunc();
// Output: "Outer Function Variable"
```

```
        console.log(outerVar);
        console.log(innerVar);
    }
    innerFunction();
}
outerFunction();
// Output: "Global Variable"
// Output: "Outer Function
Variable"
// Output: "Inner Function
Variable"
```

➔ **Importance of Closure**

- Closure is an important concept in JavaScript because it allows for data privacy and encapsulation, which are essential principles in software development.
- In JavaScript, a closure is created when a function is defined inside another function, and the inner function has access to the outer function's variables and parameters, even after the outer function has returned. This means that the inner function can continue to use and manipulate the values of those variables, without those values being accessible from outside the closure.
- Closures are important because they allow for the creation of private variables and functions that can only be accessed by the functions within the closure. This helps to prevent unintended manipulation of data by other parts of the program, improving overall code reliability and maintainability.
- Additionally, closures are often used in JavaScript to create callback functions and event handlers, which are essential for many web applications. Callback functions allow one function to be called when another function has finished executing, while event handlers are used to respond to user actions such as mouse clicks or key presses.
- Overall, closures are an important tool in JavaScript development because they provide a way to create private variables and functions, which can help to prevent bugs and improve code reliability, while also enabling the creation of powerful features such as callback functions and event handlers.

➔ API (Different way to fetch the data from public API)

- **API: (Application programming interface):** A programmer writing an application program can make a request to the Operating System using API (using a graphical user interface or command interface). It is a set of routines, protocols and tools for building software and applications. It may be any type of system like a web-based system, operating system or database System.

    - **How an API Work**
        - A set of accepted rules that specify how programmes or computers communicate with one another is known as an API. Across an application and the web server, APIs act as an intermediary layer to manage data transit between systems.

- **Different Method to use API –**

- **fetch() method:** The fetch() method is a built-in JavaScript method that is used to make HTTP requests to a server and fetch data. It returns a Promise that resolves with the response to the request. You can then use the response to extract the data you need.

    - **Key Points for Fetch method -**

- **fetch()** - This is a JavaScript API that allows you to make HTTP requests to a server and receive a response.
- **Method** - The method specifies the type of request being made. The two most common methods are "GET" and "POST". "GET" requests retrieve data from the server, while "POST" requests submit data to the server.
- **URL** - The URL (Uniform Resource Locator) specifies the location of the server and the resource being requested.
- **Headers** - Headers provide additional information about the request, such as the content type or authorization credentials.
- **Body** - The body is the payload of the request. In a "GET" request, the body is not used. In a "POST" request, the data is sent in the request body.
- **Response** - The response is the data returned by the server in response to the request. This can include text, HTML, XML, JSON, or any other format.
- **Promises** - fetch() returns a promise, which is an object representing the eventual completion or failure of the request. You can use the promise to handle the response and any errors that may occur.
- **Async/Await** - Async/await is a way of handling promises in a more synchronous-looking fashion, making the code easier to read and understand.

```
// Using fetch Method
fetch('https://dummyjson.com/comments')
.then((response)=> response.json())
.then((data)=> console.log(data))
.catch((err) => console.log(err))
```

◆ **XMLHttpRequest (XHR) object:** The XMLHttpRequest (XHR) object is an older way to fetch data from an API. It allows you to make asynchronous requests to a server and retrieve data. However, it is not as easy to use as the fetch() method and requires more code.

- **Key Points for XML**

- **XMLHttpRequest (XHR)** - This is a JavaScript API that allows you to make HTTP requests to a server and receive a response.
- **Method** - The method specifies the type of request being made. The two most common methods are "GET" and "POST". "GET" requests retrieve data from the server, while "POST" requests submit data to the server.
- **URL** - The URL (Uniform Resource Locator) specifies the location of the server and the resource being requested.
- **Headers** - Headers provide additional information about the request, such as the content type or authorization credentials.
- **Data** - The data is the payload of the request. In a "GET" request, the data is appended to the URL as a query string. In a "POST" request, the data is sent in the request body.
- **Response** - The response is the data returned by the server in response to the request. This can include text, HTML, XML, JSON, or any other format.
- **Callback function** - A callback function is a function that is executed when the response is received. This allows you to process the response and update the page accordingly.

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://dummyjson.com/comments');
xhr.onload = function(){
    if(xhr.status === 200){
        console.log(xhr.responseText);
    }else{
        console.log(`Error: ${xhr.status}`);
    }
}
xhr.send();
```

- **Axios library:**
  - Axios is a popular JavaScript library that is used to make HTTP requests from a web browser or Node.js. It simplifies the process of making HTTP requests and provides features like interceptors, request cancellation, and automatic data transformation.
  - Because of this is the javascript Library to fetch the data from API so need to import from CDN.

  - **Key Point for Axios**

  - **axios** - This is a popular JavaScript library for making HTTP requests to a server and receiving a response.
  - **Method** - The method specifies the type of request being made. The two most common methods are "GET" and "POST". "GET" requests retrieve data from the server, while "POST" requests submit data to the server.
  - **URL** - The URL (Uniform Resource Locator) specifies the location of the server and the resource being requested.
  - **Headers** - Headers provide additional information about the request, such as the content type or authorization credentials.
  - **Data** - The data is the payload of the request. In a "GET" request, the data is appended to the URL as a query string. In a "POST" request, the data is sent in the request body.
  - **Response** - The response is the data returned by the server in response to the request. This can include text, HTML, XML, JSON, or any other format.
  - **Promises** - axios returns a promise, which is an object representing the eventual completion or failure of the request. You can use the promise to handle the response and any errors that may occur.
  - **Async/Await** - Async/await is a way of handling promises in a more synchronous-looking fashion, making the code easier to read and understand.
  - **Interceptors** - Interceptors are functions that can be used to intercept requests and responses before they are handled by axios. This allows you to modify the request or response, or to handle errors in a centralized way.

- Need to a CDN Script Link : **<script src="https://unpkg.com/axios@1.1.2/dist/axios.min.js"></script>**

```
axios.get('https://dummyjson.com/posts')
.then((response) => console.log(response.data))
.catch((err) => console.log(err))
```

- **jQuery library:**
  - The jQuery library includes a method called $.ajax() that is used to make HTTP requests to an API. It provides a simple syntax for making requests and handling responses.
  - This is library of JavaScrip so need to add CDN link externally

  - **Key Point for jQuery library -**

  - **jQuery.ajax()** - This is a method in the jQuery library that allows you to make HTTP requests to a server and receive a response.
  - **Method** - The method specifies the type of request being made. The two most common methods are "GET" and "POST". "GET" requests retrieve data from the server, while "POST" requests submit data to the server.
  - **URL** - The URL (Uniform Resource Locator) specifies the location of the server and the resource being requested.
  - **Headers** - Headers provide additional information about the request, such as the content type or authorization credentials.
  - **Data** - The data is the payload of the request. In a "GET" request, the data is appended to the URL as a query string. In a "POST" request, the data is sent in the request body.
  - **Response** - The response is the data returned by the server in response to the request. This can include text, HTML, XML, JSON, or any other format.
  - **Deferred Object** - jQuery.ajax() returns a Deferred object, which is an object representing the eventual completion or failure of the request. You can use the Deferred object to handle the response and any errors that may occur.
  - **Promises** - jQuery.ajax() also supports Promises, which is an alternative way of handling asynchronous operations in JavaScript.
  - **Callbacks** - Callbacks are functions that are executed when the response is received. This allows you to process the response and update the page accordingly.

  - CDN link: **<script src="https://code.jquery.com/jquery-3.6.3.min.js"></script>**

```
$.ajax({
    url: 'https://dummyjson.com/posts',
    method: 'GET',
    success : function (response){
        console.log(response);
    },
    error : function (err) {
        console.log(err);
    }
});
```

➔ Asynchronous in JavaScript

- Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

- Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

  - **API Call - Public API or Rest API**
  - **setTimeout() and setInterval()**
  - **Call back function etc.**

- Now I am going to explain Asynchronous operation in JavaScript.
- **Call Back Function:** you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a callback function.
- A callback function is a function that is passed as an argument to another function and is then executed by that function. Callback functions are commonly used in asynchronous programming, where a function is called and continues executing while waiting for a response, and then the callback function is executed when the response is received.

- **Example with Code:**

```
function greet(name, callBack){
    console.log("Hi" + ' ' + name);
    callBack();
}
function callMe(){
    console.log("I am Callback Function...");
}
// Passing function as an argument
greet();
```

➔ **Benefit of Callback Function:**

- The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.
  - In this example, we are going to use the setTimeout() method to mimic the program that takes time to execute , such as data coming from the server.

- **Example with code**

```
function greet(){
    console.log("Callback Function...");
}
function sayName(gre){
    console.log('Welcome to'+' '+gre);
}
// Call Back Function
setTimeout(greet, 2000);
sayName('PrepBytes')
```

- ➔ **Drawback of Callback Function:** The biggest problem with callbacks is that they do not scale well for even moderately complex asynchronous code. The resulting code often becomes hard to read, easy to break, and hard to debug.

    - ◆ **Example with code**

```
function AlphaFunc(alpha, timeout){
        setTimeout(() => {
            console.log(alpha);
        }, timeout);
    }
function Abc(){
    AlphaFunc('A', 2000)
    AlphaFunc('E', 1000)
    AlphaFunc('I', 3000)
    AlphaFunc('O', 6000)
    AlphaFunc('U', 4000)
}
Abc();
// O/P - E, A, I, U, O
```

    - ◆ Here, My task is to print vowels in Sequential Manner, but the output is different. So, this is the drawback of Callback Function. For this we need to use Callback Hell.

- ➔ **Callback Hell:** The phenomenon which happens when we nest multiple callbacks within a function is called a callback hell. The shape of the resulting code structure resembles a pyramid and hence callback hell is also called the “**pyramid of the doom**”. It makes the code very difficult to understand and maintain.

- ➔ **Example with code**

```
// Write a function and Print the Vowel in given Time.
function AlphaFunc(alpha, timeout, callBack){
    setTimeout(() => {
        console.log(alpha);
        callBack()
    }, timeout);
}
function Abc(){
    AlphaFunc("A", 2000, () => {
        AlphaFunc("E", 3000, ()=> {
            AlphaFunc("I", 1000, ()=> {
                AlphaFunc("O", 6000, ()=> {
                    AlphaFunc("U", 5000,()=>{})
                })
            })
        })
    })
}
Abc();
// O/P: A, E, I, O, U
```

➔ **Drawback of Callback function**

- Callback hell is a common issue in JavaScript programming that occurs when a sequence of asynchronous functions or operations are nested within each other as callbacks. This can lead to code that is difficult to read, understand, and maintain. Some of the drawbacks of callback hell include:

    - **Code complexity:** Asynchronous operations often require multiple callbacks to handle errors and handle data returned from previous operations. This can result in code that is difficult to follow and understand.
    - **Debugging difficulties:** When code is nested inside callbacks, debugging can be challenging because it can be hard to determine which function is causing the issue.
    - **Maintenance challenges:** Callback hell can lead to code that is difficult to maintain over time. As the code base grows, it can become increasingly challenging to modify and add new features.
    - **Code duplication:** Nested callbacks can lead to duplicated code, which can increase the size of the codebase and make it harder to maintain.
    - **Lack of readability:** Callback hell can make code difficult to read, especially for developers who are new to the codebase. This can result in longer development cycles and increased development costs.

➔ Why Promise required

- A Callback is a great way when dealing with basic cases like minimal asynchronous operations. But when you are developing a web application that has a lot of code, then working with Callback will be messy. This excessive Callback nesting is often referred to as **Callback hell**.

- To Deal with such cases we have Promises instead of Callbacks.

➔ Promise In JavaScript -

- The Promise Object represents the eventual completion or failure of an asynchronous operation and its resulting value.
- It takes in two parameters resolve and reject, if we are getting a success response then we will use resolve to give response, if error comes then we will use reject to give error.
- It provides two functions **.then** and **.catch.** If you want to access response use **.then**, if you want to handle error use **.catch** function.
- The Promise represents the completion of an asynchronous operation. It returns a single value based on the operation being rejected or resolved.

    - There are mainly **three** stages of the Promise, which are shown below:
    - **Pending:** It is the initial state of each Promise. It represents that the result has not been computed yet.
    - **Fulfilled:** It means that the operation has completed.
    - **Rejected:** It represents a failure that occurs during computation.

    - ➔ Once a Promise is fulfilled or rejected, it will be immutable. The Promise() constructor takes two arguments that are a rejected function and a resolve function. Based on the asynchronous operation, it returns either the first argument or second argument.

- ➔ **There are Two way to Create Promise in JavaScript -**
    - ◆ **Using the Promise Constructor -** The Promise constructor takes a single argument, which is a function that defines the operation that the promise represents. This function takes two arguments, resolve and reject, which are functions that can be called to fulfill or reject the promise.

    - ◆ **Example with Code**

```
const promiseFunc = new Promise((resolve, reject) => {
    resolve(10);
    reject(13);
})
promiseFunc
.then((x) => console.log(x))
.catch((err)=> console.log(err));
```

- ❖ So by using Promise, How we can handle Callback hell example, the code is as below.

    - ◆ **Example with Code**

```
function resolvePromise(alpha, timeout){
           return new Promise((resolve, reject) => {
                      setTimeout(() => {
                                 console.log(alpha)
                             resolve("Promise is Resolved Successfully...");
                       }, timeout);
           })
      }
      function vowelAlpha(){
           resolvePromise('A', 2000)
           .then(()=> resolvePromise("E", 3000))
           .then(()=> resolvePromise("I", 4000))
           .then(()=> resolvePromise("O", 2000))
           .then(()=> resolvePromise("U", 7000))
      }vowelAlpha();
      //      O/P - A, E, I, O, U
```

This is the Example of Promise Chaining. Where we are using more than one .then method in a single function.

- ➔ **Async and Await in JavaScript -**
    - ◆ An async keyword will be added to a function when you want that function to platform in an asynchronous way in JS.

- For this asynchronous behavior we have to write the await keyword in the line where we want the code to hold.
- **Note:** If the async keyword is not added in the function then we cannot write await in the function.

```
function resolvePromise(alpha, timeout){
           return new Promise((resolve, reject) => {
                setTimeout(() => {
                     console.log(alpha)
                resolve("Promise is Resolved Successfully...");
                }, timeout);
           })
     }
     async function vowelAlpha(){
           await resolvePromise('A', 2000) // By using await keyword
           await resolvePromise("E", 3000)
           await resolvePromise("I", 4000)
           await resolvePromise("O", 2000)
           await resolvePromise("U", 7000)
     }
     vowelAlpha();
     // O/P - A, E, I, O, U
```

➔ Execution Context in JavaScript

- An execution context is the environment in which a piece of code is executed. It consists of the variables, functions, and objects that are available at a particular time during the execution of the code.

- **Type of Execution Context:**
    - **Global Execution Context (GEC)**
    - **Function Execution Context (FEC)**

    - **Global Execution Context (GEC) –** The global execution context is the default context that is created when JavaScript code is executed. It represents the environment in which the top-level code is executed. The global execution context includes the global object (window in browsers, global in Node.js), the this keyword, and any other global variables and functions.

    ➔ **Step How Global Execution Context works**
    - Firstly, it creates a global object where it is for Node.js and Window object for the browsers.
    - Secondly, reference the Windows object to 'this' keyword.
    - Create a memory heap in order to store variables and function references.
    - Then it stores all the function declarations in the memory heap area and the variables in the GEC with initial values as 'undefined'.

    - **Function Execution Context (FEC) –** When a function is called, a new execution context is created for that function. This context includes the function's arguments, local variables, and any nested functions or scopes. Each function call creates a new execution context, which is added to the execution stack.

    ➔ **Phase of the JavaScript Execution Context**

- **Creation Phase:** In this phase, the JavaScript engine creates the execution context and sets up the script's environment. It determines the values of variables and functions and sets up the scope chain for the execution context.
  - In the creation phase, variables will be stored in Key and Value pairs.

- **Execution phase:** In this phase, the JavaScript engine executes the code in the execution context. It processes any statements or expressions in the script and evaluates any function calls.

➔ **Code Example of Memory Allocation and Code Execution.**

Code	Creation Phase / Memory Allocation	Execution Phase / Code Execution

`var a = 10;`
`function call(num){`
`    result = num * num;`
`    return result`
`}`
`let call1 = call(5)`
`let call2 = call(a)`
`console.log(call1)`
`console.log(call2)`
	`a : undefined`
	
	`call : {...}`
	`call1 : undefined`
	`call2 : undefined`
		`a : 10`

`Function Execution for call1`

Memory Allocation	Code Execution

`num : undefined`
`result : undefined`
	`num become 5`
	`result become 25`

`Function Execution for call2`

Memory Allocation	Code Execution

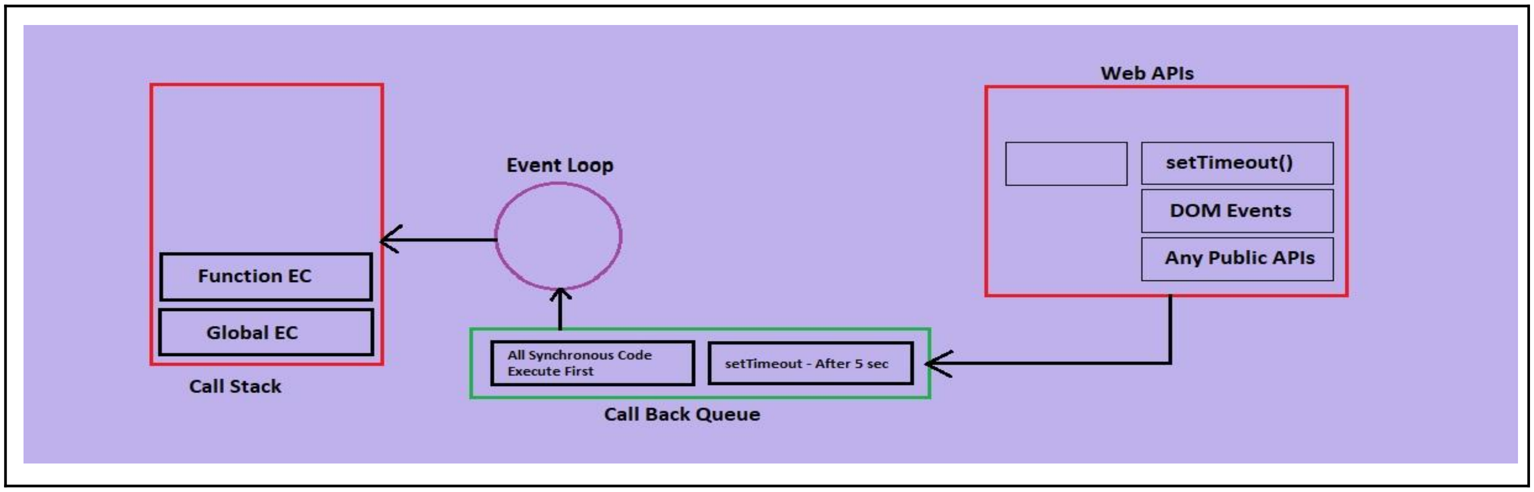
`num : undefined`
`result : undefined`
	`num become 10`
	`result become 100`

➔ After complete execution of all the code, in the Execution context, the first Function Execution context removes from call stack and the global execution context will remove from call stack and at the end of execution call stack is empty.

➔ Behavior of Execution Context when JavaScript Through Asynchronous Task in Execution Context -
- Synchronous – In javascript, Code will execute line by line. And after completion of execution of current code, then it will move to next execution.
- Asynchronous – when some Web APIs, DOM Event, setTimeout method is used, so the execution context doesn't know when the code is executed. So execution context executes all synchronous code first, and execute Asynchronous.

➔ **Term used in Execution Context -**
- **Event Loop –** The Event Loop has one simple job — to monitor the Call Stack and the Callback Queue. If the Call Stack is empty, the Event Loop will take the first event from the queue and will push it to the Call Stack, which effectively runs it.
- **Call Back Queue –** A callback queue is a queue of tasks that are executed after the current task. The callback queue is handled by the JavaScript engine after it has executed all tasks in the microtask queue.
- **Web APIs –** Web APIs is a place where the Asynchronous code waits till the time to execute.
- **Call Stack –** A Call stack is a queue of tasks, where all the javascript code will execute. **Call Stack Follow First In First Out.**

➔ Diagram of Execution Context



➔ OOPs concept in JavaScript - (Object Oriented Programming in JavaScript)

- **What is OOPs?**
  - OOP (Object-Oriented Programming) is an approach in programming in which data is encapsulated within objects and the object itself is operated on, rather than its component parts. And introduced in ES6 version.

- **There are certain features or mechanisms, which make a language Object-Oriented Programming**

Class	Object	Inheritance	Polymorphism

➔ **Class:** Classes are the special type of functions. We can define the class just like function declarations and function expressions. The JavaScript class contains various class members within a body including methods or constructor. The class is executed in strict mode. So, the code containing the silent error or mistake throws an error.

- **Class syntax contain Two components:**
  - **Class declarations –** A class can be defined by using a class declaration. A class keyword is used to declare a class with any particular name. According to JavaScript naming conventions, the name of the class always starts with an uppercase letter.

  - Classes are blueprints of an Object. A class can have many Objects because the class is a template while Objects are instances of the class or the concrete implementation.

Example with code -

```
class Employee  {
//Initializing an object
    constructor(id,name){
      this.id=id;
      this.name=name;
    }
//Declaring method
    detail(){
```

```
Output:

101 PrepBytes
102 CollegeDekho
```

```
        console.log(this.id+" "+this.name);
    }
}
//passing object to a variable
var e1=new Employee(101,"PrepBytes");
var e2=new Employee(102,"CollegeDekho");
e1.detail(); //calling method
e2.detail(); // Calling Method
```

- Class declaration is not a part of JavaScript Hoisting. So, it is required to declare the class before invoking it.
- Re-declaring class - A class can be declared once only. If we try to declare class more than one time, it throws an error.

- **Class expressions –** Another way to define a class is by using a class expression. Here, it is not mandatory to assign the name of the class. So, the class expression can be named or unnamed. The class expression allows us to fetch the class name. However, this will not be possible with class declaration.

- **Unnamed Class Expression:** The class can be expressed without assigning any name to it.

➔ **Example with code**

```
var emp = class {
    constructor(id, name) {
      this.id = id;
      this.name = name;
    }
  };
  let x = new emp(10023555, "Shubham");
console.log(x);
```
	```
	// Output :
	// emp { id: 10023555, name:
	'Shubham' }
	```

- **Objects:** JavaScript is an Object based Language. Everything is an object in JavaScript.
  - An Object is a unique entity that contains **properties** and **methods**.
  - For example "a car" is a real-life Object, which has some characteristics like color, type, model, and horsepower and performs certain actions like driving. The characteristics of an Object are called Properties in Object-Oriented Programming and the actions are called methods. An Object is an instance of a class. Objects are everywhere in JavaScript, almost every element is an Object whether it is a function, array, or string.

- The object can be created in two ways in JavaScript:

  - **Object Literal**

```
const car = {
    color: "Light White",
    type: "SUV",
    model: "XUV 700",
```
	```
	// Output :
	{
	color: 'Light White',
	type: 'SUV',
	model: 'XUV 700',
	```

```
    horsepower: "2400 CC"
}
console.log(car);
```

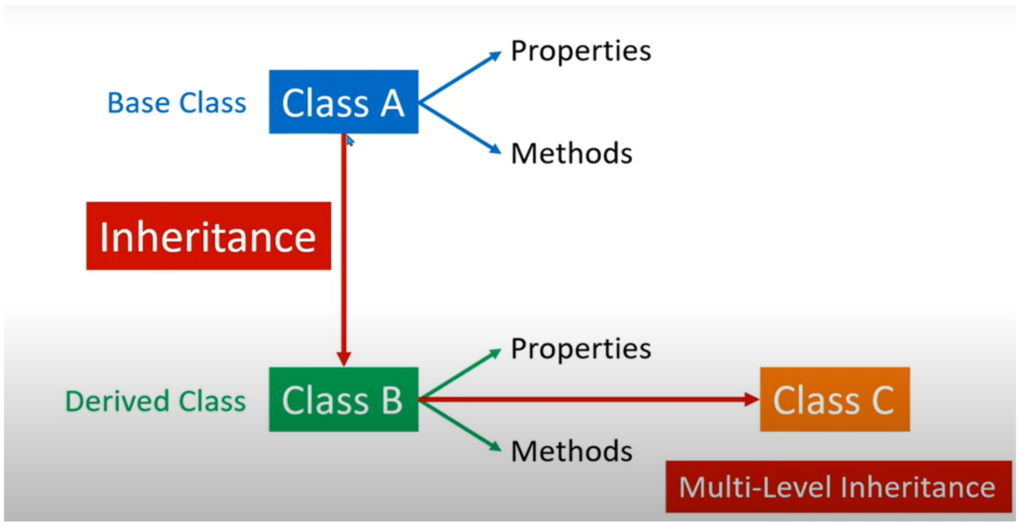
```
horsepower: '2400 CC'
}
```

- **Object Constructor –** You need to create a function with arguments. Each argument value can be assigned in the current object by using this keyword.
- The **this keyword** refers to the current object.

```
function emp(id,name,salary){
    this.id=id;
    this.name=name;
    this.salary=salary;
}
const e=new emp(10023555,"Admin",9530);
console.log(e.id+" "+e.name+" "+e.salary);
```

```
// Output :
10023555
Admin
9530
```

- **Inheritance:** It is a concept in which some properties and methods of an Object are being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Objects inherit Objects i.e. certain features (property and methods) of one object can be reused by other Objects.



➔ **Example**

```
class Person{
    constructor(name){
        this.name = name;
    }
    print(){
        console.log(`Hi My Name is ${this.name}`)
    }
```

```
// Output: student
    { name:
'Shubham',
        id: 10023555,
        status:
'Complete'
}
```

```
}
class student extends Person{
    constructor(name, id, status){
        // super keyword for calling the above
        super(name);
        this.id = id,
        this.status = status
    }
    print(){
        console.log(`Name is ${this.name} id:
${this.id}, status: ${this.status} : ${super.name} `)
    }
}
const Per = new Person("Milan")
const stu = new student("Shubham", 10023555, "Complete")
console.log(stu, Per);
```

```
// Output:
Person
{ name: 'Milan' }

// we define a Person
Object with certain
properties and
methods and then we
inherit the Person
Object in the Student
Object and use all
the properties and
methods of the Person
Object as well as
define certain
properties and
methods for the
Student Object.
```

The super keyword is used to refer to the immediate parent class's instance variable.

- **Note:** The Person and Student objects both have the same method (i.e print()), this is called Method Overriding. Method Overriding allows a method in a child class to have the same name(polymorphism) and method signature as that of a parent class.

- **Polymorphism:** Polymorphism is one of the core concepts of object-oriented programming languages. Polymorphism means the same function with different signatures is called many times. In real life, for example, a boy at the same time may be a student, a class monitor, etc. So a boy can perform different operations at the same time. Polymorphism can be achieved by method overriding and method overloading.

```
class A{
    display(){
      console.log("A is invoked");
    }
}
class B extends A{
}
var b=new B();
b.display();
```

```
// Output : A is invoked
```

➔ Session Storage, Local Storage and Cookies

- **Session storage, local storage are the property of the window object but cookies are the property of the document.**

- **Local Storage –** Local storage is a way for web applications to store key-value pairs locally within a user's browser. It provides a simple and easy-to-use API for web developers to store and retrieve data.
  - **Some of the Key Points of Local Storage**
  - Local storage is specific to a particular domain, meaning that data stored by one domain cannot be accessed by another domain.
  - Local storage is persistent and will remain available even after the user closes their browser and returns to the web application.
  - Local storage can only store strings, so any other data types will need to be converted before being stored.
  - Local storage has a maximum size limit of around 5-10MB, depending on the browser.
  - **Important Method of Local Storage**
    - **setItem:** This method is used to store a key-value pair in local storage. The key parameter is a string that represents the key for the item, while the value parameter can be any data type that can be serialized to a string.

```
localStorage.setItem("username", "PrepBytes");
```

    - **getItem:** This method is used to retrieve the value associated with a given key in local storage. The key parameter is a string that represents the key for the item.

```
const username = localStorage.getItem("username");
console.log(username); // outputs "PrepBytes"
```

    - **removeItem:** removeItem(key): This method is used to remove a key-value pair from local storage. The key parameter is a string that represents the key for the item.

```
localStorage.removeItem("username");
```

    - **Clear:** This method is used to remove all key-value pairs from local storage.

```
localStorage.clear();
```

Ex:

```
//Create a Array and Store some data
const arrayData = [89, 90, 96, 93, 90];
// store the data in local storage using setItem
localStorage.setItem("marks", arrayData);
// get the data from local storage using getItem
let data = localStorage.getItem("marks");
console.log(data)
// Output: 89, 90, 96, 93,90
localStorage.removeItem('marks')
// Output: Remove all item of marks
// Create other object in local storage
localStorage.setItem('username', "Shubham");
localStorage.getItem('username');
localStorage.clear();
// Output: remove all the key and Value in local storage.
```

- **Session Storage:** Session storage is a mechanism that allows you to store key/value pairs in the user's web browser, for the duration of a session. A session is typically defined as the time between when a user opens a web page and when they close it. When the user closes the browser, all data stored in session storage is deleted.
    - **Some of the Key Points of Session Storage**
    - Session storage is specific to a particular browser tab or window. If the user opens a new tab or window, the session storage will be unique to that tab or window.
    - Data stored in session storage can only be accessed by JavaScript that is running on the same domain as the page that stored the data. This is to prevent malicious websites from stealing sensitive data.
    - The data stored in session storage is limited to a certain amount of space, which varies between browsers. You should be careful not to store too much data in session storage, or your website may become slow or unresponsive.
    - The data in session storage persists until the user closes the browser window or tab, or until the session expires. You can also manually clear the data using JavaScript.

    - **Important Method of session storage**
        - **setItem(key, value):** This method sets a key-value pair in the session storage object. The **key** parameter is the name of the key, and the value parameter is the value associated with the key. If the key already exists, its value will be updated.
        - **getItem(key):** This method retrieves the value associated with a given key in the session storage object. The key parameter is the name of the key.
        - **removeItem(key):** This method removes the key-value pair associated with a given key from the session storage object. The key parameter is the name of the key.
        - **clear():** This method removes all key-value pairs from the session storage object.

```
//Create a Array and Store some data
const companyName = ["PrepBytes", "CollegeDekho", "GirnarSoft"];
// store the data in session storage using setItem
sessionStorage.setItem("company", companyName);
// get the data from session storage using getItem
let data = localStorage.getItem("company");
console.log(data);
// Output: "PrepBytes", "CollegeDekho", "GirnarSoft"
sessionStorage.removeItem('company');
// Output: this data remove from session storage
sessionStorage.clear();
//Output: remove all data from session storage
```

- **Cookies:** Cookies are small pieces of data that are stored on the user's device by the web server, and they are sent back to the server with every request made by the user. Cookies are used to store user preferences, login information, shopping cart items, and other types of data that help websites remember user behavior and provide a personalized browsing experience.
    - **Some of Key point of Cookies**

- Cookies are typically used to store small amounts of data that can be accessed by the server.
- Cookies are stored on the client-side, usually in the browser's cache or file system.
- Cookies have an expiration date, after which they will be deleted automatically.
- Cookies can be accessed and modified by both the client-side and server-side scripts.
- Cookies can be used to track user behavior across different websites, which can raise privacy concerns.

- **Important Method of Cookies**
  - **document.cookies –** This property is used to read or write a cookie value. The **document.cookie** property is used to read or write a cookie value. When reading, document.cookie returns a string containing all the cookies for the current domain. When writing, document.cookie is set to a string that represents the cookie to be stored.

```
document.cookie = "username = Shubham Kumar";
```

  - **encodeURIComponent() –** This function is used to encode a cookie value before setting it.
  - The **encodeURIComponent()** function is used to encode a cookie value before setting it. This function is necessary because cookie values can contain special characters, such as semicolons, commas, and spaces, which can interfere with cookie parsing.

```
let key = "a;;";
let value = "7654";
document.cookie = `${encodeURIComponent(key)} = ${encodeURIComponent(value)}`
console.log(document.cookie);
// Output: a%3B%3B=7654
```

  - **decodeURIComponent() –** This function is used to decode a cookie value after reading it.
  - The **decodeURIComponent()** function is used to decode a cookie value after reading it. This function is necessary because cookie values are typically encoded using **encodeURIComponent()**.

```
let key = "a;;";
let value = "7654";
// encodeURIComponent encode the key
document.cookie = `${encodeURIComponent(key)} = ${encodeURIComponent(value)}`
console.log(document.cookie)
// Output: a%3B%3B=7654
// decodeURIComponent encode the key
document.cookie1 = `${decodeURIComponent(document.cookie)}`
console.log(document.cookie1);
// Output: a;; = 7654
```

➔ Error in JavaScript

- **Error –** In programming, an error is an unexpected event or condition that prevents the program from executing correctly. In JavaScript, errors can occur due to syntax errors, runtime errors, and logic errors.

◆ **There are Four type of JavaScript errors**

- **Syntax Error –** A syntax error in JavaScript occurs when the code is written in an incorrect format or structure. This means that there is an error in the way that the code is written and the JavaScript interpreter cannot understand it.

`let name = 'PrepBytes'`
`console.log(name+);`
	`console.log(name+);`
	`^`
	`SyntaxError: Unexpected token ')'`
	`Because compiler not understand name+`

- **Reference Error –** A reference error is a type of error that occurs in JavaScript when you try to reference a variable or function that has not been defined or is not in scope.

`let subject = "JavaScript";`
`console.log(course);`
	`console.log(course);`
	`^`
	`ReferenceError: course is not defined`

- **Type Error –** In JavaScript, a type error occurs when a value is of the wrong type and cannot be operated on or used in a certain context. For example, if you try to perform a mathematical operation on a string or access a property on an undefined variable, a type error will be thrown.

`let num = 1;`
`let str = num.toUpperCase();`
`console.log(str)`
	`let str = num.toUpperCase();`
	`^`
	`TypeError: num.toUpperCase is not a function`

- **Range Error –** A Range Error is a type of error that occurs in JavaScript when a value is not within an expected range. This error typically occurs when attempting to perform an operation that requires a value within a specific range, such as accessing an array index that does not exist, or passing a value that is outside of the acceptable range for a function argument.

`const arr = [1,2,3,4,5];`
`console.log(arr[10]);`
	`Output : undefined`
	`arr have 5 index only index 0 to 4 but try to access index 10`

★ Project Detail -

S No.	Name Of Project	Host and Repo URL	
1	Countdown App	*Repo URL*	*Host URL*
2	Calculator	*Repo URL*	*Host URL*
3	Dice Game	*Repo URL*	*Host URL*
4	Dictionary (API)	*Repo URL*	*Host URL*
5	JS Blog Project	Repo URL	Host URL
6	Tic Tac Toe Game	*Repo URL*	*Host URL*