



# JavaScript Programming language

## E-Book



**Er. Rajesh Prasad(B.E, M.E)**  
Founder: RID Organization

- **RIDORGANIZATION**यानि Research, InnovationandDiscoveryसंस्था जिसका मुख्य उद्देश्य हैं आने वाले समय में सबसे पहले **NEW(RID, PMS & TLR)**की खोज, प्रकाशन एवं उपयोग भारतकी इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW(RID, PMS & TLR)**के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इसRID**संस्था की स्थपना किया हूँ।
- Research, Innovation&Discovery में रुचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इसRID**संस्थासे जुड़ें एवं अपने बुधिद, विवेक एवं प्रतिभा से दुनियां को कुछ नई (**RID, PMS & TLR**)कीखोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें।

“त्वक्सा JavaScript प्रोग्रामिंग लैंगेज के इस ई-पुस्तक में आप JavaScript से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे।। मुझेआशाहैकिइसई-पुस्तककोपढ़नेकेबादआपकेज्ञानमेंवृद्धिहोगीऔरआपकोकंप्यूटरविज्ञानकेबारेमेंऔरअधिकज्ञाननेमेंरुचिहोगी”

“In this E-Book of JavaScript Programming language you will learn all the basic concepts related to JavaScript. I hope after reading this E-Book your knowledge will be improve and you will get more interest to know more thing about computer Science”.

### Online & Offline Class:

**Web Development, Java, Python Full Stack Course, Data Science Training, Internship & Research**

करने के लिए Message/Call करें. 9892782728 E-Mail\_id: [ridorg.in@gmail.com](mailto:ridorg.in@gmail.com)

Website: [www.ridbharat.com](http://www.ridbharat.com)

## **RIDहमें क्यों करना चाहिए?**

### **(Research)**

अनुसंधान हमें क्यों करना चाहिए ?

#### **Why should we do research?**

1. नई ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (ToSocial progress)
4. विकास को बढ़ावा देने( To promote development)
5. तकनीकी और व्यापार में उन्नति(Toadvances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

### **(Innovation)**

नवीनीकरणहमें क्यों करना चाहिए ?

#### **Why should we do Innovation?**

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(ToBenefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

### **(Discovery)**

खोजहमें क्यों करना चाहिए?

#### **Why should we doDiscovery?**

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(ToDiscovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास(To develop the country's science & technology)

### ❖ Research(अनुसंधान):

- अनुसंधानएक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिजाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

### ❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

### ❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

**नोट :** अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

### सुविचार:

1.	समस्याओं का समाधान करने का उत्तम मार्ग हैं। → शिक्षा, RID, सहयोग, प्रतिभा, एकता एवं समाजिक कार्य-
2.	एक इंसान के लिए जरूरी हैं। → गोठी, कपड़ाइज्जत और सम्मान, रोजगार, शिक्षा, मकान,
3.	एक देश के लिए जरूरी हैं। → संस्कृतिएकता, भाषा, सभ्यता-, आजादीसंविधान एवं अखंडता,
4.	सफलता पाने के लिए होना चाहिए। → लक्ष्यप्रतिबद्धता, शक्ति-इच्छा, त्याग, प्रतिभा एवं सतता
5.	मरने के बाद इंसान छोड़कर जाता है। → शरीरकर्म एवं विचार, नाम, परिवार-घर, धन-अन्
6.	मरने के बाद इंसान को इस धरती पर याद किया जाता है उनके

→ नाम समर्पण एवं कर्मों से-सेवा, विचार, दान, काम, ...

### आशीर्वाद (बड़े भैया जी)



Mr. RAMASHANKAR KUMAR

### मार्गदर्शन एवं सहयोग



Mr. GAUTAM KUMAR



...सोच है जिनकी नई कुछ कर दिखाने की, खोज है रीड संस्था को उन सभी इंसानों की...

“अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रुचि रखते हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहते हैं एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहते हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें”॥ धन्यवाद || **Er. Rajesh Prasad (B.E, M.E)**



S. No:	Topic Name	Page No:
1	What is JavaScript & Features of javascript ?	4
2.	History and application of javascript	5
3	Javascript comments	6
4	How to run javascript code	7
5	Javascript output	11
6	Input statement	18
7	Number system	28
7	Javascript Identifiers And Variable	31
8	Difference between var const and let	35
9	Javascript operators	39
10	Data types	49
11	Non-primitive data types	51
12	Base Conversion in javascript	56
13	Control statement	59
14	Conditional statements	60
15	Loop statement	68
16	Functions	80
17	Arrow function	90
18	Function invocation(call)	92
19	Higher-order functions	96
20	Object	103
21	Array	117
22	String	132
23	This keyword	144
24	Hoisting	146
25	Math in javascript	151
26	Number object	155
27	Advantages & Disadvantages Of Oops	158
28	Browser object model(bom)	179
29	Navigator object	188
30	Javascript screen object	190
31	Cookies	192
32	Javascript debugging	195
33	Javascript promises	198
34	Document object model(dom)	204
35	Finding html objects	227
36	Javascript HTML DOM Elements & Events	230
37	Regular expressions	220
38	Javascript Validation	264
39	Web apis, AJAX & JSON	267-274
40	Exception Handling in js	276
41	Local Storage in js	280
42	Javascript Connection With database((Mysql) & Mongo DB )	291- 296
43	Important question and answer & what is RID?	297-302

# **JAVASCRIPT**

- JavaScript is a versatile, high-level programming language primarily used to create interactive and dynamic content on websites. It enables client-side scripting, allowing web pages to respond to user actions and update content without reloading.
- JavaScript is a dynamic programming language. It is lightweight and most commonly used as a part of web pages. It is an interpreted programming language.
- It is platform independent and object-oriented programming language.
- JavaScript is used to modify the HTML content.
- In HTML, JavaScript code is inserted between <script>.....</script> tags.

## **FEATURES OF JAVASCRIPT**

- **High-level Language:** it is a high-level programming language, means Human readable.
- **Interpreted Language:** JavaScript is typically executed by a web browser's JavaScript engine or a server-side runtime environment like Node.js. It doesn't require compilation before execution, making it a scripting language.
- **Dynamic Typing:** it is dynamically typed, allowing variables to change types at runtime.
- **Weak Typing:** JavaScript is also weakly typed, which means it performs type coercion automatically when operations involve different data types.
- **Client-Side Scripting:** JavaScript is primarily used for client-side scripting in web development. It runs directly in web browsers and can manipulate the Document Object Model (DOM) to change the content and behaviour of web pages dynamically.
- **Cross-Platform:** it is supported by all major web browsers, making it a cross-platform language.
- **Object-Oriented:** it is an object-oriented programming language
- **First-Class Functions:** JavaScript treats functions as first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned from functions.
- **Closures:** JavaScript supports closures, which allow inner functions to access variables from their outer containing functions even after the outer function has finished executing. Closures are crucial for maintaining state and data encapsulation.
- **Asynchronous Programming:** JavaScript excels in handling asynchronous operations, such as AJAX requests, timers, and event-driven programming. Call-backs, Promises, and the async/await syntax make it easier to manage asynchronous code.
- **Modularity:** JavaScript allows you to organize code into modules, improving code maintainability and reusability. The ES6 module system, introduced in ECMAScript 2015, provides a standardized way to define and import/export modules.
- **Extensible:** JavaScript can be extended through libraries, frameworks, and external APIs. A rich ecosystem of third-party libraries and tools exists to enhance JavaScript's capabilities.
- **Standardized:** JavaScript is standardized through the ECMAScript specification, which defines the language's core features.
- **Security:** it has built-in security features to protect against malicious code execution.
- **Community and Ecosystem:** JavaScript has a vast and active developer community, contributing to its continuous growth and innovation.
- **Scalability:** With the advent of server-side JavaScript via Node.js, JavaScript can be used for both front-end and back-end development, making it a suitable choice for building scalable, full-stack applications.



## **HISTORY OF JAVASCRIPT**

- Sir Tim Berners Lee introduced web in early 1990's he introduced a language called HTML.
- In 1993, Mosaic, the first popular web browser, came into existence. In the year 1994, Netscape was founded by Marc Andreessen. He realized that the web needed to become more dynamic. Thus, a 'glue language' was believed to be provided to HTML to make web designing easy for designers and part-time programmers. Consequently, in **1995, the company recruited Brendan Eich** intending to implement and embed Scheme programming language to the browser. But, before Brendan could start, the company merged with Sun Microsystems for adding Java into its Navigator so that it could compete with Microsoft over the web technologies and platforms. Now, two languages were there: Java and the scripting language. Further, Netscape decided to give a similar name to the scripting language as Java's. It led to 'JavaScript'. Finally, in May 1995, Marc Andreessen coined the first code of JavaScript named 'Mocha'. Later, the marketing team replaced the name with 'Live Script'. But, due to trademark reasons and certain other reasons, in December 1995, the language was finally renamed to 'JavaScript'. From then, JavaScript came into existence.

## **APPLICATION OF JAVASCRIPT**

- **Web Development:** Primarily used for creating interactive, dynamic web applications.
- **Front-End Development:** Core to front-end development, with frameworks like React, Angular, and Vue.js for building complex UIs.
- **Single-Page Applications (SPAs):** Used in SPAs to update content dynamically without reloading the page.
- **Mobile App Development:** Tools like React Native enable JavaScript-based mobile app development for iOS and Android.
- **Backend Development:** Node.js allows JavaScript to be used for backend, enabling full-stack development with one language.
- **Game Development:** With HTML5 and libraries like Phaser, JavaScript supports web-based game creation.
- **Web APIs:** Enables interaction with external data sources like social media, weather services, and more.
- **Web Animation:** Used with HTML5 <canvas> and CSS to create animations and effects.
- **Browser Extensions:** JavaScript builds extensions for browsers like Chrome and Firefox.
- **Data Visualization:** Libraries like D3.js and Chart.js help create interactive charts and data visualizations.
- **Real-Time Applications:** Powers real-time applications such as chat, gaming, and collaborative tools.
- **Cross-Platform Desktop Apps:** Electron and NW.js allow JavaScript-based desktop applications across platforms.
- **Internet of Things (IoT):** JavaScript builds dashboards for controlling and monitoring IoT devices.
- **Serverless Computing:** Used in platforms like AWS Lambda for serverless applications and microservices.
- **Automation:** Works with headless browsers (e.g., Puppeteer) for web scraping, testing, and form automation.
- **Machine Learning:** Libraries like TensorFlow.js enable running ML models directly in the browser.

# JAVASCRIPT COMMENTS

- The JavaScript comments are meaningful way to deliver message. It is used to add information about the code, warnings or suggestions
- **Types of JavaScript Comments**
- There are two types of comments in JavaScript.

  1. Single-line Comment
  2. Multi-line Comment

- **JavaScript Single line Comment**
- It is represented by double forward slashes (//). It can be used before and after the statement.

**Example:**

```
<script>
    var a=10
    var b=30
    var c=a+b
    console.log(c); // this console
    document.write(c) // this is document.write data
</script>
```

➤ **JavaScript Multi line Comment:**

- It can be used to add single as well as multi line comments. So, it is more convenient.
- /\* It is multi line comment.
- It will not be displayed \*/

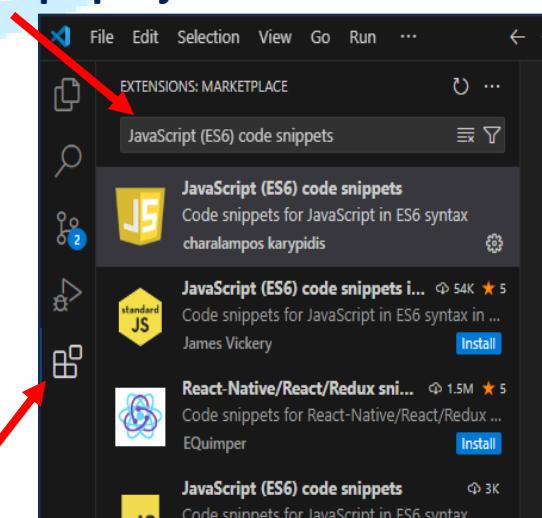
**Example:**

```
<script>
    /* console.log("welcome to T3 skills center")
    let a = 20
    let b = 10 */
    let c = a + b
    console.log(c)
    var d = 30
    console.log("value of d=", d)
</script>
```

**These extensions streamline development, improve code quality, and enhance productivity in JavaScript projects.**

- **ESLint:** Ensures consistent code by checking syntax and style.
- **Prettier:** Auto-formats code to a standard style, often with ESLint.
- **JavaScript (ES6) Snippets:** Provides ES6 snippets for faster coding.
- **Debugger for Chrome:** Debugs JavaScript directly in Chrome.
- **Path Intellisense:** Autocompletes file paths for easier imports.
- **Live Server:** Launches a local server with live reloading.
- **GitLens:** Enhances Git integration with commit and change tracking.
- **Jest:** Provides real-time test feedback in VSCode.
- **Code Spell Checker:** Identifies typos in code for better readability.
- **npm Intellisense:** Autocompletes npm modules in imports.

**Note:** Install all these extensions in VS Code or any other text editor.



## **HOW TO RUN JAVASCRIPT CODE**

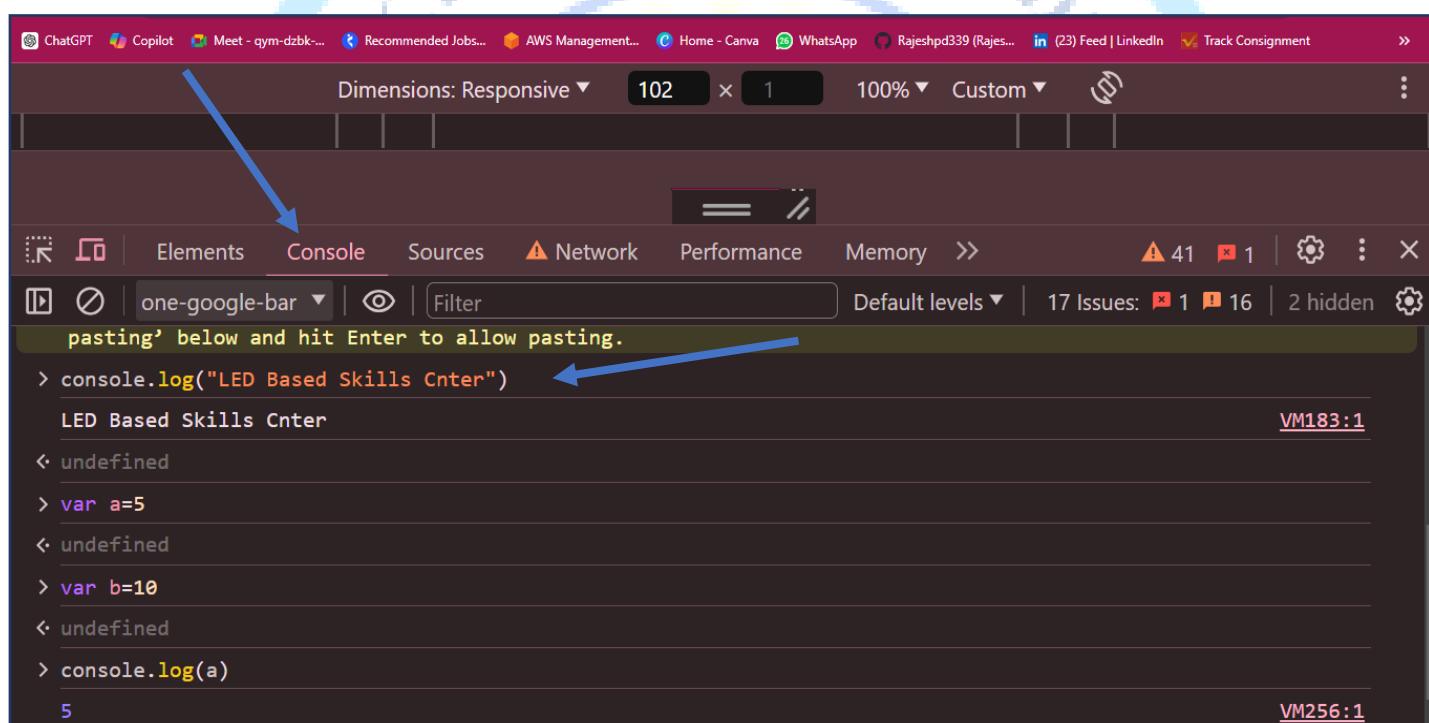
- Running JavaScript code can be done in various environments, each suited for different tasks. Here are all the main ways to run JavaScript code:
- 1) **Browser Console:** Run JavaScript directly in the browser's developer tools console.
  - 2) **HTML File:** Embed code within <script> tags in an HTML file and open it in a browser.
  - 3) **Node.js:** Write code in a .js file and run it using node filename.js command in terminal.

### **1. In a Web Browser Console:**

- Web browsers have built-in consoles where you can run JavaScript code.

#### **Steps:**

- Open your web browser (e.g., Chrome, Firefox, Edge).
- Press F12 or right-click on the page and select "Inspect" to open Developer Tools.
- Go to the "Console" tab.
- Type or paste your JavaScript code and press Enter.



## 2. By Using Script tag:

- **Internal JavaScript** is written within the `<script>` tags, as shown in your example.
- **External JavaScript** is linked using the `src` attribute in a `<script>` tag, referencing an external `.js` file, like in your example (`<script src="raj.js"></script>`).
- **Inline JavaScript** is written within HTML tag attributes (like `onclick`, `onload`, etc.) to perform specific actions, such as handling events.

### Example:

```
<!DOCTYPE html> <html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head> <body>
    <h1>Welcome to RID Bharat .</h1>
    <!-- Inline JavaScript (Event Handler) -->
    <button onclick="alert('Hello, Good Morning!')">Click Me</button> <br>
    <!-- Apply Internal JavaScript -->
    <script>
        document.write("Good Morning", "<br>")
        let a = 10; let b = 20;
        let sum = a + b;
        document.write("Sum of two numbers = ", sum, "<br>")
    </script> <!-- Apply External JavaScript -->
    <script src="raj.js"></script>
</body> </html>
```

The screenshot shows a code editor interface with the following details:

- Explorer View:** Shows files in the project: RAJ (index.html, model.html, mohan.html, pm.jpg), JS (raj.js), and RID.html.
- RID.html Content:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head> <body>
    <h1>Welcome to RID Bharat .</h1>
    <!-- Inline JavaScript (Event Handler) -->
    <button onclick="alert('Hello, Good Morning!')">Click Me</button> <br>
    <!-- Apply Internal JavaScript -->
    <script>
        document.write("Good Morning", "<br>")
        let a = 10;
        let b = 20;
        let sum = a + b;
        document.write("Sum of two numbers = ", sum, "<br>")
    </script> <!-- Apply External JavaScript -->
    <script src="raj.js"></script>
</body> </html>
```
- raj.js Content:**

```
let n1=50
let n2=100
let sum3=n1+n2
document.write("sum of two number=",sum3)
```
- Browser Preview:** Shows the output of RID.html. It displays "Welcome to RID Bharat ." and a button labeled "Click Me". Below the button, it says "Good Morning" and "Sum of two numbers = 30". A modal dialog box appears with the text "127.0.0.1:5500 says Hello, Good Morning!" and an "OK" button.



### 3. Using Node.js

- Node.js is a runtime environment that allows you to run JavaScript on the server-side or directly on your machine.

#### ❖ install Node.js and run your first JavaScript program

##### Step 1: Install Node.js

1. Go to the [official Node.js website](#).
2. Download the **LTS (Long Term Support)** version for your operating system.
3. Run the installer and follow the on-screen instructions.

##### Step 2: Verify Node.js Installation

1. Open a terminal or command prompt.
2. Type the following command to verify installation:

#### ➤ node -v and npm -v

This will show the installed Node.js version if it's correctly installed.

##### Step 3: Create Your First JavaScript File

1. Open a text editor (like VSCode or Notepad).
2. Create a new file and name it app.js.
3. Add your JavaScript code, for example:

➤ `console.log('Welcome To, RID Bharat Org.');`

##### Step 4: Run the JavaScript File

1. In your terminal, navigate to the folder where app.js is located.
2. Run the file using Node.js:

#### ➤ node app.js

3. You should see the output:

Welcome To, RID Bharat Org.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

about_Execution_Policies at https://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ npm install -g npm
+ ~~~~
+ CategoryInfo          : SecurityError: () [], PSNotSupportedException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\hp\Desktop\raj> History restored
v22.11.0
PS C:\Users\hp\Desktop\raj> npm -v
disabled on this system. For more
information, see about_Execution_Policies at https://go.microsoft.com/fwlink/?LinkID=135
170.
At line:1 char:1
+ npm -v
+ ~~~~
+ CategoryInfo          : SecurityError: () [], PSNotSupportedException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\hp\Desktop\raj> node -v
v22.11.0
  
```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\hp\Desktop\raj> Get-ExecutionPolicy
>>
Restricted
PS C:\Users\hp\Desktop\raj> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
>>
PS C:\Users\hp\Desktop\raj> Get-ExecutionPolicy
>>
RemoteSigned
PS C:\Users\hp\Desktop\raj> npm -v
10.9.0
PS C:\Users\hp\Desktop\raj> npm update
up to date in 2s
  
```

The error you're encountering is due to the PowerShell execution policy on your system, which is blocking the execution of scripts like npm.ps1.

#### Steps to Fix:

1. Open PowerShell as Administrator:
  - o Press Win + X and select Windows PowerShell (Admin) or Command Prompt (Admin) if using an older version of Windows.
2. Check the Current Execution Policy:  
**Get-ExecutionPolicy**

This will show the current execution policy, which is likely set to Restricted or AllSigned.

3. Change the Execution Policy: To allow scripts to run, you can set the execution policy to RemoteSigned or Unrestricted (which is less restrictive). You can use one of the following commands:

**Set-ExecutionPolicy RemoteSigned - Scope CurrentUser**



## Steps:

- Download and install Node.js from the Node.js official website.
- Open a command line (Command Prompt, PowerShell, or terminal).
- Create a .js file with your JavaScript code in any text editor like vs code .
- Run the file using Node.js.

```

changed 13 packages in 30s

24 packages are looking for funding
  run `npm fund` for details
PS C:\Users\hp\OneDrive\Desktop\demo3> npm -v
10.8.0
PS C:\Users\hp\OneDrive\Desktop\demo3> node -v
v18.17.1
PS C:\Users\hp\OneDrive\Desktop\demo3> npm install -g npm
  
```

OUTPUT

```

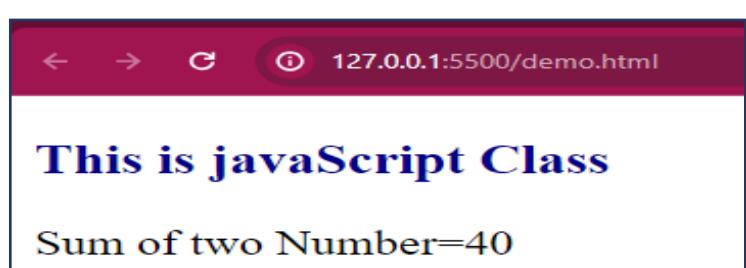
Hello, World!
[Done] exited with code=0 in 1.042 seconds
[Running] node "c:\Users\hp\OneDrive\Desktop\demo3\helo.js"
Hello, World!
welcome to T3 Skills Center
[Done] exited with code=0 in 0.532 seconds
  
```

### Create file demo.html

```

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<h3 style="color: darkblue;">This is JavaScript Class</h3>
<script>
  var a=10
  var b=30
  var c=a+b
  console.log(c);
  document.write("Sum of two Number=",c)
</script>
</body></html>
  
```

❖ Clear the Terminal Screen  
Command>> Clear-Host  
✓ Screen will be clear



# JAVASCRIPT OUTPUT

- In JavaScript we can "display" data in this following ways:
- 1) Writing into the browser console, using `console`
  - 2) Writing into an HTML element, using `innerHTML`.
  - 3) Writing into the HTML output using `document.write()`.
  - 4) Writing into an alert box, using `window.alert()`.
  - 5) Writing into the HTML input tag output using `.value=+variabl_name;`

## 1. Using `console.log()`

- For debugging purposes, you can call the `console.log()` method in browser.

### Example:

```
console.log("Welcome to T3 skills Center")
let Name="RID Org"
console.log("Organization Name="+Name)
let a=3
console.log(a)
```

### OutPut:

```
[Running] node "c:\Users\hp\Desktop\demo.js"
Welcome to T3 skills Center
Organization Name=RID Org
3
```

- ❖ **Console:** `console` object provides different methods for logging messages to the browser console. Each method is used for different purposes:

### 1. `console.log():`

- **Purpose:** General logging of information.
- **Usage:** Use this method to print regular messages or information to the console.
- **Example:** `console.log("RID Organization")` - This will print "RID Organization" to the console.

### 2. `console.info():`

- **Purpose:** Informational messages.
- **Usage:** Use this method to log informational messages that are not errors but might be useful to understand the flow or state of the application.
- **Example:** `console.info("Skills Center")` - This will print "Skills Center" to the console, often with an "info" icon.

### 3. `console.error():`

- **Purpose:** Error messages.
- **Usage:** Use this method to log error messages. These messages indicate something went wrong in the code.
- **Example:** `console.error("Error Message")` - This will print "Error Message" to the console, often with a red background or an error icon to highlight the severity.

### 4. `console.warn():`

- **Purpose:** Warning messages.
- **Usage:** Use this method to log warning messages. These messages are less severe than errors but indicate potential issues or important notices.
- **Example:** `console.warn("WIT Center")` - This will print "WIT Center" to the console, often with a yellow background or a warning icon.

## 2. Display data in js By Using innerHTML

- To access an HTML element, you can use the document.getElementById(id) method.
- id attribute defines the HTML element. The innerHTML property defines HTML content:  
**Example:**

### There are following js method are used for the access the HTML elements

1. **getElementById()**: Gets a single element by its unique id.
2. **getElementsByName()**: Gets a live HTMLCollection of elements by class name.
3. **getElementsByTagName()**: Gets a live HTMLCollection of elements by tag name.
4. **getElementsByName()** method is used to select **all elements** in the DOM that have a specified name attribute.
5. **querySelector()**: Gets the first element that matches a CSS selector.
6. **querySelectorAll()**: Gets a NodeList of all elements that match a CSS selector.

#### Example: Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <h1 class="c c1"></h1>
  <h2 class="c c2"></h2>
  <p>hi</p>
  <p>hello</p>
  <pre>...</pre>
  <div name="value"></div>
  <div name="value"></div>
  <h3 class="cc1">...</h3>
  <h3 id="dd1"></h3>
  <p class="qll">...</p>
  <p class="qll">...</p>
  <p class="qll">...</p>
  <p id="Res"></p>
  <script src="demo.js"></script>
</body> </html>
```

#### Example: demo.js

```
document.getElementsByClassName("c")[0].innerHTML="RID"
document.getElementsByClassName("c")[1].innerHTML="Research"
document.getElementsByClassName("c1")[0].innerHTML="Innovation"
document.getElementsByTagName("p")[0].innerHTML="Discovery"
document.getElementsByTagName("p")[1].innerHTML="TIP"
document.getElementsByName("value")[0].innerHTML="Training"
document.getElementsByName("value")[1].innerHTML="Internship"
document.querySelector(".cc1").innerHTML="Placement"
document.querySelector("#dd1").innerHTML="Workshop"
document.querySelectorAll(".qll")[0].innerHTML="LED Based Skills Center"
document.querySelectorAll(".qll")[1].innerHTML="Learning Earning
Development"
let a = 20
let b = 19
let c = a + b
document.getElementById("Res").innerHTML="sum= "+c
```

**Note:[0]:** This index is used to select the first (and possibly only) element in that HTMLCollection.

**RID**

**OutPut**

**Research**

**Innovation**

**Discovery**

**TIP**

...

**Training  
Internship**

**Placement**

**Workshop**

**LED Based Skills Center**

**Learning Earning Development**

...

**sum= 39**



❖ **To assign multiple classes to a single HTML element,**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Multiple Classes Example</title>
</head>
<body>
<h1 id="d1" class="class1 class2">Hello World</h1>
<script>
// Alternatively, using querySelector
document.querySelector(".class1.class2").innerHTML = "Welcome to T3 Skills Center";
</script>
</body>
</html>
```

**Example for Multiple Class and multiple index value**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
</head> <body>
<h1 class="c1"></h1>
<p class="n n1"></p>
<h1>RID:</h1>
<ol type="1">
<li class="h1"></li>
<li class="h1"></li>
<li class="h1"></li>
</ol>

<script>
document.getElementsByClassName("c1")[0].innerHTML = "Welcome to T3 Skills Center";
document.querySelector(".n.n1").innerHTML = "Research";
var selector=document.getElementsByClassName("h1")
selector[0].innerHTML="Research"
selector[1].innerHTML="Innovation"
selector[2].innerHTML="Discovery"
</script>
</body> </html>
```

**Output:**

Welcome to T3 Skills Center  
Research  
RID:  
1. Research  
2. Innovation  
3. Discovery

**Note:**

- The `getElementsByClassName("class1")` method returns a collection of all elements with the class `class1`.
- [0] accesses first element, [1] accesses second element, & [2] accesses third element.
- Each element's `innerHTML` property is updated accordingly.
- If you try to access an index that doesn't exist (e.g., [3] in this case), you won't get an error, but the result will be `undefined` since there is no fourth element.

- ❖ **querySelector():-** querySelector is a method in JavaScript that allows you to select the **first element** in the document that matches a **CSS selector** (or group of selectors). This method is more versatile and powerful than older methods like **getElementById**, **getElementsByClassName**, or **getElementsByTagName** because it allows you to use complex CSS selectors to target elements more precisely.

## Syntax:

**document.querySelector(css selector);**

- **selector:** A **string** containing one or more CSS selectors. This can be any valid CSS selector, such as an ID (#id), class (.class), or element (div), or even more complex selectors like div > .class, input[type="text"], etc.
- **Returns:** The **first element** that matches the given CSS selector. If no elements match, it returns null.
- ❖ **Example Usage of querySelector**

### 1. Selecting by ID:

let element = document.querySelector("#myId");

- This selects the first element with the ID myId.

### 2. Selecting by Class:

let element = document.querySelector(".myClass");

- ✓ This selects the **first element** with the class myClass.

### 3. Selecting by Element Tag:

let element = document.querySelector("h1");

- ✓ This selects the **first <h1> element** in the document.

### 4. Using Complex Selectors:

let element = document.querySelector("div > .myClass");

- ✓ This selects the **first element** with class myClass that is a direct child of a <div>.

### 5. Selecting by Attribute:

let element = document.querySelector("input[type='text']");

- **Summary:**

- **querySelector** is a versatile and powerful method for selecting elements based on **CSS selectors**. It's useful because you can target specific elements in a highly flexible and consistent way.
- It's preferred in modern web development for its simplicity, flexibility, and the ability to handle complex queries.
- For selecting multiple elements that match a selector, you can use **querySelectorAll**, which returns a **NodeList**

#### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Display Data in One Element</title>
</head>
<body>
<!-- Multiple h2 tags -->
<h1 class="raj raj1">Hi</h1>
<h1 class="m"></h1>
<h2 class="m"></h2>
<h2>Heading 1</h2>
<h2>Heading 2</h2>
<h2>Heading 3</h2>
<!--
<p class="q1">hello</p>
<p class="q1">kya</p>
<p class="q1">name</p>
<script>
// let a = document.querySelector(".q1");
let a = document.querySelectorAll(".q1");
a.innerHTML = "ram ram bhai sahab"
</script>-->
<p class="q1">hello</p>
<p class="q1">kya</p>
<p class="q1">name</p>
<script>
// Select all elements with the class 'q1'
let ab = document.querySelectorAll(".q1");

// Loop through each element in the NodeList and change its content
ab.forEach(function(element) {
    element.innerHTML = "ram ram "; // Modify content for each element
});
</script>
<script>
let abc = document.querySelector(".raj.raj1").innerHTML = "hello every to this"
let a = document.getElementsByClassName("m");
a[1].innerHTML = "RID"
a[0].innerHTML = "RID Bharat"

// Access all <h2> tags on the page
let b = document.getElementsByTagName("h2");
// Display data in the first <h2> element (index 0)
b[1].innerHTML = " TIR"; // Set the content of the first h2 tag
</script>
</body>
</html>
```



### 3. Using document.write()

- For testing purposes, it is convenient to use document.write()
- Using document.write() after an HTML document is loaded, will delete all existing HTML: The document.write() method should only be used for testing.

#### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<script>
let a=10
let b=20
let c=a+b
document.write("Welcome to RID Organization"+<br>)
document.write("Welcome to T3 Skills Center"+<br>+<br>)
document.write("value of a="+a)+"<br>"
document.write("value of b="+b+"<br>")
document.write("Sum of two Number="+c)
</script>
</body>
</html>
```

#### Output:

Welcome to RID Organization

Welcome to T3 Skills Center

value of a=10 value of b=20

Sum of two Number=30

### 4. Using window.alert()

- You can use an alert box to display data:
- You can skip the window keyword.
- In JavaScript, the window object is the global scope object. This means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional:

#### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<script>
let a=10
let b=20
let c=a+b
window.alert("Welcome to T3 Skills Center")
window.alert("value of a="+a)
window.alert("Sum of Two Number="+c)
alert("value of b="+b)
</script>
</body>
</html>
```

**Note:** window.alert () inside we can separate elements with comma ,

Example: window.alert("Sum of two Number=",c) this is wrong

Example: window.alert("sum of two number=" +c) this is correct

## 5. Display output in input box

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>by using the form tag take input</title>
</head>
<body>
<form>
<label>Number-1</label>
<input type="text" id="num-1"><br><br>
<label>Number-2</label>
<input type="text" id="num-2"><br><br>
<button type="button" id="click" onclick="add()">Add</button> <br><br>
<label>Result</label>
<input type="text" id="res">
</form>
<button onclick="alert('hello every ')">click</button>
<script>
function add(){ // document.querySelector("#num-1").value="hello"
a=document.querySelector("#num-1").value
b=document.querySelector("#num-2").value
c=a+b
n=parseInt(a)
m=parseInt(b)
sum=n+m
document.getElementById("res").value+=sum
document.querySelector("#num-1").value=""
document.querySelector("#num-2").value=""
```

Number-1	40
Number-2	50
<input type="button" value="Add"/>	
Result	<input type="text"/>
<input type="button" value="click"/>	



# **INPUT STATEMENT IN JS**

- Input statement is used take data from user.
- There is various method to take user input in JavaScript.
  1. By using Prompt
  2. By using HTML Form Elements
  3. By using Event Listeners
  4. By using Inline Event Handlers
  5. By using confirm and alert

## **1. Using prompt:**

- The prompt function displays a dialog box that prompts the user for input and returns the input as a string.
- While the prompt function is not natively available in Node.js, you can achieve similar functionality using external libraries. One popular library for this purpose is prompt-sync, which allows synchronous prompting of user input in Node.js.

### **❖ Install prompt-sync:**

- Open your terminal or command prompt and install prompt-sync:
  - **npm install prompt-sync**

**Example-1:** const prompt = require('prompt-sync')();

```
let num1 = parseInt(prompt("Enter the Number-1: "));  
let num2 = parseInt(prompt("Enter the Number-2: "));  
let sum = num1 + num2;  
console.log("Sum of Two Numbers =", sum);
```

### **OutPut:**

PS C:\Users\hp\OneDrive\Desktop\demo3>npm i prompt-sync

PS C:\Users\hp\OneDrive\Desktop\demo3> node demo3.js

Enter the Number-1: 10

Enter the Number-2: 20

Folder Name

Sum of Two Numbers = 30

File Name

### **Example-3:**

```
const prompt = require('prompt-sync')();  
let name = prompt("Enter your Name: ");  
console.log("Name=", name);
```

### **Ex-5:<!DOCTYPE html>**

```
<html lang="en"><head>  
<meta charset="UTF-8">  
<title>Document</title>  
</head><body>  
<input type="text" id="print">  
<script>  
let name = prompt("Enter your name");  
document.getElementById("print").value = name;  
</script>  
</body></html>
```

### **Example-2:**

```
const prompt=require("prompt-sync")()  
let a=prompt("Enter the value of a=")  
let b=prompt("Enter the value of b=")  
let sum=Number(a)+Number(b)  
console.log("sum of two num="+sum)
```

### **Example-4:**

```
const prompt = require('prompt-sync')();  
let value = parseFloat(prompt("Enter float value: "));  
console.log("Value =", value);
```

```
<!DOCTYPE html><html lang="en">  
<head><meta charset="UTF-8">  
</head><body>  
<label for="d5"></label>  
<input type="text" id="d5"><br>  
<script>let num=prompt("Enter the value of a=")  
let num1=prompt("enter the value of number-2=")  
let t1=typeof(num), let t2=typeof(num)  
document.write(t1+"<br>")  
document.write(t1+"<br>")  
let sum=Number(num)+Number(num1)  
document.getElementById("d5").value="sum =" +sum  
</script>  
</body></html>
```



## 2. By using HTML Form Elements:

- You can create form elements in HTML and access their values using JavaScript.

### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title></head>
<body>
<form id="userForm">
<label for="username">Username:</label>
<input type="text" id="username" name="username"><br><br>
<label for="email">Email:</label>
<input type="email" id="email" name="email"><br><br>
<input type="button" value="Submit" onclick="processForm()"><br><br>
<p id="successful"></p>
<label for="user_name">Your User Name:</label>
<input type="text" id="user_name"><br><br>
<label for="Email_Id">Your Registered Email Id:</label>
<input type="email" id="Email_Id">
</form>
<script>
function processForm() {
var username = document.getElementById("username").value;
var email = document.getElementById("email").value;
document.getElementById("successful").innerHTML = "Registration Successful";
document.getElementById("user_name").value = username;
document.getElementById("Email_Id").value = email; }
</script>
</body></html>
```

### How to empty the input box

```
document.getElementById("username").value=""
document.getElementById("email").value=""
```

← → ⌛ 127.0.0.1:5500/index.html

Username: RID Organization

Email: ridorg.in@gmail.com

Submit

Registration Successful

Your User Name: RID Organization

Your Registered Email Id: ridorg.in@gmail.com

### Example: When <button type="submit"> will be

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head> <body>
<form>
    <label>Number-1</label>
    <input type="text" id="d1"><br><br>
    <label>Number-2</label>
    <input type="text" id="d2"><br><br>
<button type="submit" onclick="add(event)">Add</button>
    <label>Result</label>
    <input type="text" id="res">
</form>
```

Number-1 30

Number-2 9

Add

Result 39

```
<script>
function add(event){
    event.preventDefault()
    let a=document.getElementById("d1").value
    let b=document.getElementById("d2").value
    sum= parseInt(a)+parseInt(b)
    document.getElementById("res").value=sum
}</script> </body> </html>
```

### Example: When <button type="submit"> will be

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
<form>
    <label>Number-1</label>
    <input type="text" id="d1"><br><br>
    <label>Number-2</label>
    <input type="text" id="d2"><br><br>
<button type="button" onclick="add()">Add</button>
    <label>Result</label>
    <input type="text" id="res">
</form>
```

Number-1 30

Number-2 9

Add

Result 39

```
<script>
function add(){
    let a=document.getElementById("d1").value
    let b=document.getElementById("d2").value
    sum= parseInt(a)+parseInt(b)
    document.getElementById("res").value=sum
}</script> </body></html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Web RID</title>
</head> <body>
<form>
    <label>User Name</label>
    <input id="d1" type="text"><br><br>
    <label>Password</label>
    <input id="d2" type="text"><br><br>
<input type="submit" onclick="sub(event)" value="Submit">
    <h1 class="c1"></h1>
    <h2 class="c2"></h2>
</form>
```

```
<script>
function sub(event) {
    event.preventDefault(); // Prevent form submission
    let a = document.getElementById("d1").value;
    let b = document.getElementById("d2").value;
    document.querySelector(".c1").innerHTML = a;
    document.querySelector(".c2").innerHTML = b;
}
</script> </body></html>
```

User Name rid339

Password 339

Submit

rid339

339



### 3. by using Event Listeners:

- event listeners are mechanisms that allow you to detect and respond to events that occur on an HTML element, such as a **mouse click, keyboard press, form submission**, or any other interaction.
- You can handle user input events such as change, input, or submit.

#### Example for event Listener

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Web RID</title>
</head>
<body>
  <form id="sub">
    <label>User Name</label>
    <input id="d1" type="text" autocomplete="off"><br><br>
    <label>Password</label>
    <input id="d2" type="text" autocomplete="off"><br><br>
    <!-- <input type="submit"> -->
    <button type="submit">Submit</button>
    <!-- <button type="button">Submit</button> it will not work for the eventListener -->
    <h1 class="c1"></h1>
    <h2 class="c2"></h2>
  </form>
  <script>
    document.getElementById("sub").addEventListener("submit",function(event){
      event.preventDefault(); // Prevent form submission
      // Get input values
      let a = document.getElementById("d1").value;
      let b = document.getElementById("d2").value;
      // Update the content of .c1 and .c2
      document.querySelector(".c1").innerHTML = a;
      document.querySelector(".c2").innerHTML = b;
      // for clear the screen
      document.getElementById("d1").value;
      document.getElementById("d2").value;
    })
  </script>
</body> </html>
```

browser might still save form data due to its autocomplete feature. To ensure that the browser does not save the data you input, you can add the autocomplete attribute to your form and input elements and set it to "off".

#### Example:

```
<form id="form1" autocomplete="off">
  <label>number-1</label>
```

User Name

Password

**rid339**

**339**



## Example: cursor should go one input box to another input box through Enter bottom press from keyword

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.6.0/css/all.min.css"
integrity="sha512-Kc323vGBEqzTmouAECnVceyQqyqdsSiqlQISBL29aUW4U/M7pSPA/gEUZQqv1cwx4OnYxTxve5UMg5GT6L4JJg=="
crossorigin="anonymous" referrerpolicy="no-referrer" />
<style>
    button{
        width: 40px;
        height: 40px;
        font-size: 25px;
    }
</style>
</head>
<body>
<form>
    <label>Number-1</label>
    <input id="d1" type="text" placeholder="Enter the number-1" autocomplete="off"><br><br>
    <label>Number-2</label>
    <input id="d2" type="text" placeholder="Enter the number-2" autocomplete="off"><br><br>
    <button id="addbtm" type="button" onclick="add()"><i class="fa-solid fa-plus"></i></button>
    <button id="subbtm" type="button" onclick="sub()"><i class="fa-solid fa-minus"></i></button>
    <button id="mulbtm" type="button" onclick="mul()"><i class="fa-solid fa-xmark"></i></button>
    <button id="divbtm" type="button" onclick="div()"><i class="fa-solid fa-divide"></i></button>
    <button id="molbtm" type="button" onclick="mol()"><i class="fa-solid fa-percent"></i></button> <br><br>
    <label>Result</label>
    <input type="text" id="d3" >
    <script src="abhi.js"></script>
</form>
</body>
</html>
```

Number-1

Number-2

Result



```
// Function to add two numbers
function add() {
    let a = document.getElementById("d1").value;
    let b = document.getElementById("d2").value;
    let sum = parseFloat(a) + parseFloat(b);
    document.getElementById("d3").value = sum;
    document.getElementById("d1").value = "";
    document.getElementById("d2").value = "";
}

// Function to divide two numbers
function div() {
    let x = document.querySelectorAll("#d1")[0].value;
    let y = document.querySelectorAll("#d2")[0].value;
    let quotient = parseFloat(x) / parseFloat(y);
    document.querySelector("#d3").value = quotient;
    document.getElementById("d1").value = "";
    document.getElementById("d2").value = "";
}

// Event listener for cursor navigation between
// input fields and buttons
document.getElementById("d1").addEventListener
("keypress", function (addmsg) {
    if (addmsg.key === "Enter") {
        addmsg.preventDefault();
        document.getElementById("d2").focus();
    }
});
document.getElementById("d2").addEventListener
("keypress", function (msg) {
    if (msg.key === "Enter") {
        msg.preventDefault();
        document.getElementById("addbtm").focus();
    }
});
// Keyboard navigation for buttons
document.getElementById("addbtm").addEventListener("keydown", function (msg1) {
    if (msg1.key === "ArrowRight") {
        msg1.preventDefault();
        document.getElementById("subbtm").focus();
    }
});
document.getElementById("subbtm").addEventListener("keydown", function (msg1) {
    if (msg1.key === "ArrowLeft") {
        msg1.preventDefault();
        document.getElementById("addbtm").focus();
    }
});
```

```
// Function to subtract two numbers
function sub() {
    let n = document.getElementById("d1").value;
    let m = document.getElementById("d2").value;
    let sub = parseFloat(n) - parseFloat(m);
    document.getElementById("d3").value = sub;
    document.getElementById("d1").value = "";
    document.getElementById("d2").value = "";
}
```

```
// Function to multiply two numbers
function mul() {
    let x = document.querySelectorAll("#d1")[0].value;
    let y = document.querySelectorAll("#d2")[0].value;
    let mul = parseFloat(x) * parseFloat(y);
    document.querySelector("#d3").value = mul;
    document.getElementById("d1").value = "";
    document.getElementById("d2").value = "";
}
```

```
// Function to calculate the remainder (modulo) of two
// numbers
function mol() {
    let x = document.querySelectorAll("#d1")[0].value;
    let y = document.querySelectorAll("#d2")[0].value;
    let rem = parseFloat(x) % parseFloat(y);
    document.querySelector("#d3").value = rem;
    document.getElementById("d1").value = "";
    document.getElementById("d2").value = "";
}
```

```
if (msg1.key === "ArrowRight") {  
    msg1.preventDefault();  
    document.getElementById("mulbtm").focus();  
}  
});  
document.getElementById("mulbtm").addEventListener("keydown", function (msg2) {  
    if (msg2.key === "ArrowLeft") {  
        msg2.preventDefault();  
        document.getElementById("subbtm").focus();  
    }  
    if (msg2.key === "ArrowRight") {  
        msg2.preventDefault();  
        document.getElementById("divbtm").focus();  
    }  
});  
document.getElementById("divbtm").addEventListener("keydown", function (sadi) {  
    if (sadi.key === "ArrowLeft") {  
        sadi.preventDefault();  
        document.querySelector("#mulbtm").focus();  
    }  
    if (sadi.key === "ArrowRight") {  
        sadi.preventDefault();  
        document.getElementById("molbtm").focus();  
    }  
});  
document.getElementById("molbtm").addEventListener("keydown", function (sadi) {  
    if (sadi.key === "ArrowLeft") {  
        sadi.preventDefault();  
        document.getElementById("divbtm").focus();  
    }  
});  
  
// Event listener to navigate between d1 and d2 using ArrowUp and ArrowDown  
document.getElementById("d2").addEventListener("keyup", function (job) {  
    if (job.key === "ArrowUp") {  
        job.preventDefault();  
        document.getElementById("d1").focus();  
    }  
});  
document.getElementById("d1").addEventListener("keyup", function (job) {  
    if (job.key === "ArrowDown") {  
        job.preventDefault();  
        document.getElementById("d2").focus();  
    }  
});
```

#### 4. by using Inline Event Handlers:

- **Inline event handlers** are a way to directly associate event-handling JavaScript code with an HTML element. The event handler is written as an attribute inside the HTML tag, and its value contains JavaScript code to be executed when specified event occurs.

##### Example-1:

```
<button onclick="alert('Hello, World!')>Click Me</button>
```

##### Example-2:

```
<script>
    function greet(name) {
        alert('Hello, ${name}!');
    }
</script>
<button onclick="greet('Ankit')>Greet</button>
```

##### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<form>
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" onchange="Change()">
    <h1 id="d1"></h1>
</form>
<script>
function Change() {
    // Get the value of the username input field
    var username = document.getElementById("username").value;
    // Update the content of the element with ID 'd1'
    document.getElementById("d1").innerHTML = username;
    // Log the username in the console
    console.log("Username=", username);
}
</script>
</body></html>
```

Username: Rid Bharat

**Rid Bharat**



**RID BHARAT**

## 5. By using confirm and alert

- While not for input, confirm and alert are often used alongside prompt for user interactions.

### Example-1:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<script>
let ans= confirm("Do you want to proceed?");
if (ans) {
alert("You chose to proceed!");
} else {
alert("You chose to cancel!");
}
</script>
</body> </html>
```

### Example-2:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
</head>
<body>
<script>
var ans = confirm("Do you want to add the two number?");
if (ans) {
let a=parselnt(prompt("Enter the first number: "))
let b=parselnt(prompt("Enter the second number: "))
let sum=a+b
document.write("Sum of two number=",sum)
alert("sum of two number="+sum)
// alert("You chose to proceed!");
} else {
alert("You chose to cancel!");
}
</script>
</body> </html>
```



## Complete Example of use prompt, form elements, and event listeners

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>User Input Example</title>
</head>
<body>
<h1 id="header"></h1>
<p class="info"></p>
<h3 class="program"></h3>
<form id="userForm">
<label for="username">Username:</label>
<input type="text" id="username" name="username"><br><br>
<label for="email">Email:</label>
<input type="email" id="email" name="email"><br><br>
<input type="submit" value="Submit">
</form>
<script>
    // Using prompt to get user input
    var orgName = prompt("Enter the organization name:");
    var tagline = prompt("Enter the tagline:");
    var program = prompt("Enter the program name:");
    // Displaying the input from prompt
    document.getElementById("header").innerHTML = "Welcome to " + orgName;
    document.getElementsByClassName("info")[0].innerHTML = tagline;
    document.getElementsByClassName("program")[0].innerHTML = program;
    // Using event listener for form submission
    document.getElementById("userForm").addEventListener("submit", function(event) {
        event.preventDefault(); // Prevent the form from submitting the traditional way
        var username = document.getElementById("username").value;
        var email = document.getElementById("email").value;
        console.log("User input from form - Username:", username);
        console.log("User input from form - Email:", email);
    });
</script>
```

## Welcome to T3 Skills Center

Coding class

**JavaScript**

Username:

Email:

# **NUMBER SYSTEM**

## **2) Binary Number:**

- Binary number system, is a base-2 numeral system that uses two symbols: **0 and 1**.
- It's the foundation of digital technology and computing.
- Binary is fundamental in modern computing because digital devices, such as computers and microcontrollers, use binary to process and store data.
- All data, including text, images, sound, and videos, is ultimately represented in binary form within these devices.
- it directly corresponds to the on and off states of electronic switches and represents information using two distinct states.
- $(0, 1) \text{ ( )}_2$  {Base or Radix}

**Example:** 0, 1, 01, 10, 1110, 10101011, 111001110101 etc.

## **3) Decimal Number:**

- The decimal number system, also known as the base-10 number system, is a positional numeral system that uses ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) to represent numbers.
- It's the most common number system used by humans in everyday life.
- The decimal system is essential for everyday arithmetic, commerce, science, and a wide range of applications.
- $(0,1.....9) \text{ ( )}_{10}$  {Base or radix}

**Example:** 0,1,2,4,5,6,7,8,9,10,11,12,13,14, 15.....

## **4) Octal Number:**

- The octal number system, also known as base-8, is a positional numeral system that uses eight symbols (0, 1, 2, 3, 4, 5, 6, and 7) to represent numbers.
- The octal system was more commonly used in computing systems that were based on multiples of 3 (as opposed to the binary system's base-2).
- However, octal has largely been replaced by hexadecimal (base-16) in modern computing due to its compatibility with binary and its more compact representation.
- $(0,1,2.....7) \text{ ( )}_8$  {Base or radix}

**Example:** 0,1,2,3,4,5,6,7,10, 11,...17,20,21.....27,30,31.....

## **5) Hexadecimal Number:**

- Hexadecimal number system, often referred to as "hex" or base-16, is a positional numeral system that uses sixteen symbols: 0-9 for values 0 to 9, and A-F (or a-f) for values 10 to 15.
- The hexadecimal system is widely used in computing and digital systems as a concise way to represent binary data and memory addresses.
- Hexadecimal is often used in programming and computer science because it provides a more compact representation of binary data, and it's easier to work with when converting between binary and other number systems.
- $(0 .....9, A,B,C,D,E,F) \text{ ( )}_{16}$  {Base or Radix}

**Example:** 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F, 1a,1b,1c .....etc



# NUMBER SYSTEM CONVERSION

- Number system conversion is the process of converting a value from one numeral system (base) to another.

## ❖ Decimal Number to Binary Number Conversion

- Question-1: Convert  $(27)_{10}$ ,  $(137)_{10}$  and  $(66)_{10}$  Decimal into binary number

1<sup>st</sup> Method

	27
2	13 ----- 1
2	6 ----- 1
2	3 ----- 0
1	----- 1

Ans. 11011

	137
2	68 ----- 1
2	34 ----- 0
2	17 ----- 0
2	8 ----- 1
2	4 ----- 0
2	2 ----- 0
1	----- 0

Ans. 10001001

	66
2	33 ----- 0
2	16 ----- 1
2	8 ----- 0
2	4 ----- 0
2	2 ----- 0
1	----- 0

Ans. 1000010

2<sup>nd</sup> Method

$2^8$	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	Ans: $(27)_{10} = (11011)_2$
256	128	64	32	16	8	4	2	1	
$(27 > 16)$	$27 =$	16	+8	8	+4	4	+2	2	
		16	+8	8	+4	4	+2	2	
		1	1	0	1	1	1	1	

## ❖ Decimal Number to Octal Number Conversion:

- Question-2: Convert  $(27)_{10}$ ,  $(294)_{10}$  and  $(137)_{10}$  Decimal into octal number

Ans.  $(33)_8$

	27
3	----- 3

Ans.  $(446)_8$

	294
8	36 ----- 6
4	----- 4

Ans.  $(211)_8$

	137
8	17 ----- 1
2	----- 1

## ❖ Decimal Number to Hexadecimal Number Conversion:

- Question-3: Convert  $(27)_{10}$ ,  $(294)_{10}$  and  $(137)_{10}$  Decimal into Hexadecimal number

Ans.  $(1B)_{16}$

	27
1	----- B

Ans.  $(126)_{16}$

	294
16	18 ----- 6
1	----- 2

Ans.  $(89)_{16}$

	137
16	8 ----- 9
2	----- 1



### ❖ **Binary to Decimal:**

1.  $(100010010)_2 = ( )_{10}$

➤ Solution:  $1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$   
 $256 + 0 + 0 + 0 + 16 + 0 + 0 + 2 + 0 = 274 \quad (274)_{10} \text{ Ans.}$

2.  $(10010)_2 = ( )_{10}$

➤ Solution:  $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$   
 $16 + 0 + 0 + 2 + 0 = 18 \quad (18)_{10} \text{ Ans.}$

3.  $(1110100)_2 = ( )_{10}$

➤ Solution:  $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$   
 $64 + 32 + 16 + 0 + 4 + 0 + 0 = 116 \quad (116)_{10} \text{ Ans.}$

### ❖ **Octal to Decimal:**

1.  $(422)_8 = ( )_{10}$

➤ Solution:  $4 \times 8^2 + 2 \times 8^1 + 2 \times 8^0$   
 $4 \times 64 + 2 \times 8 + 2 \times 1$   
 $256 + 16 + 2 = 274 \quad (274)_{10} \text{ Ans.}$

2.  $(4211)_8 = ( )_{10}$

➤ Solution:  $4 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 1 \times 8^0$   
 $4 \times 512 + 2 \times 64 + 1 \times 8 + 1$   
 $2048 + 128 + 8 + 1 = 2185 \quad (2185)_{10} \text{ Ans.}$

3.  $(333)_8 = ( )_{10}$

➤ Solution:  $3 \times 8^2 + 3 \times 8^1 + 3 \times 8^0$   
 $3 \times 64 + 3 \times 8 + 3 \times 1$   
 $192 + 24 + 3 = 219 \quad (219)_{10} \text{ Ans.}$

### ❖ **Hexadecimal to decimal:**

1.  $(422)_{16} = ( )_{10}$

➤ Solution:  $4 \times 16^2 + 2 \times 16^1 + 2 \times 16^0$   
 $4 \times 256 + 2 \times 16 + 2 \times 1$   
 $1024 + 32 + 2 = 1058 \quad (1058)_{10} \text{ Ans.}$

2.  $(4211)_{16} = ( )_{10}$

➤ Solution:  $4 \times 16^3 + 2 \times 16^2 + 1 \times 16^1 + 1 \times 16^0$   
 $4 \times 4096 + 2 \times 256 + 1 \times 16 + 1$   
 $16384 + 512 + 16 + 1 = 16913 \quad (16913)_{10} \text{ Ans.}$

3.  $(333)_{16} = ( )_{10}$

➤ Solution:  $3 \times 16^2 + 3 \times 16^1 + 3 \times 16^0$   
 $3 \times 256 + 3 \times 16 + 3 \times 1$   
 $768 + 48 + 3 = 819 \quad (819)_{10} \text{ Ans.}$

4.  $(333AB)_{16} = (3 \times 16^4) + (3 \times 16^3) + (3 \times 16^2) + (10 \times 16^1) + (11 \times 16^0) = (209835)_{10}$

5.  $(DCF39)_{16} = (13 \times 16^4) + (12 \times 16^3) + (15 \times 16^2) + (3 \times 16^1) + (9 \times 16^0) = (905017)_{10}$

6.  $(AB23F34)_{16} = (10 \times 16^6) + (11 \times 16^5) + (2 \times 16^4) + (3 \times 16^3) + (15 \times 16^2) + (3 \times 16^1) + (4 \times 16^0) = (179453748)_{10}$

# JAVASCRIPT IDENTIFIERS

- All JavaScript variables must be identified with unique names.
- These unique names are called identifiers.
- Rules of naming variables in JavaScript:
  1. special keywords shouldn't use example-if,for,let etc.
  2. Names must begin with a letter.
  3. No space allowed
  4. Name can start with underscore or alphabets
  5. Names can contain letters, digits, underscores, and dollar signs.

## ❖ Naming conventions

- 1.camelCase- ex- addOfTwoNumbers
- 2.PascalCase- ex-AddOfTwoNumbers
- 3.snake\_case- ex-add\_of\_two\_numbers

# JAVASCRIPT VARIABLE

- variable is a name of memory location.
- A Variable have four attributes.
  1. **Name:** A variable has a unique identifier, known as its name.
  2. **Data Type:** Every variable has a data type that specifies what kind of data it can hold, such as integers, floating-point numbers, characters, etc.
  3. **Value:** A variable holds a value, which can be assigned during declaration or later in the program's execution. The value can change during the program's execution.
  4. **Reference & Scope:** it help for system to identify value in memory The scope of a variable defines where in the program it can be accessed. Variables can be local to a function or block, or they can be global, accessible throughout the entire program.

## Syntax:

{ Left Side = Right Side }

Name              Value

Variable\_Name

Variable

A=3

Data Type

Typeof() or simple typeof  
console.log("Type=",typeof(339))

Reference

Amount of Space

identify value in memory

In Js, you cannot directly access or find the memory location of a variable like you can in some lower-level languages (such as C or C++) or use the id() function in Python. JavaScript abstracts away memory management for security and simplicity reasons.

Value

For this create function

➤ There are two types of variables in JavaScript:

1. local variable and
2. global variable.

➤ Rules for declaring a JavaScript variable (also known as identifiers).

- Name must start with a letter (a to z or A to Z), underscore (\_ ), or dollar( \$ ) sign.
- After first letter we can use digits (0 to 9), for example value1.
- JavaScript variables are case sensitive, for example a and A are different variables.

➤ JavaScript local variable:

- A JavaScript local variable is declared inside block or function. It is accessible within the function or block only.

**Example:**

```
<script>
function abc(){
    varx=10; // local variable
}
</script>
```

Or

```
<script>
if(10<15) {
    var y=20; // JavaScript local variable
}
</script>
```

**1.local variable Example:**

```
{
    let name="RID"
    console.log(name) //RID
}
console.log(name) // Error
```

Note: we can name variable outside block because it local variable

**2. global variable Example.**

```
let name="RID"
{
    console.log(name)
}
console.log(name)
```

## ❖ Variable Declaration

- JavaScript Variables can be declared in 4 ways:
  1. Automatically
  2. Using var
  3. Using let
  4. Using const

**Note:**; semicolon is optional in javascript

**Example:**

- a, b, and c are undeclared variables.
- They are automatically declared when first used.

**File p1.js**

```
a=6
b=9
c=a+b
console.log(c)
```

**Output:**PS D:\js rev\rajt3> node ./p1.js 15

- From the above examples you can guess:  
a stores the value 6  
b stores the value 9  
c stores the value 15



### Example using var

File: Jp.js

```
Var x=9;  
Var y=6;  
Var z=x+y;  
console.log(z)
```

**Output:** PS C:\Users\hp\OneDrive\Desktop\demo3> node jp.js 15

#### Note:

- The var keyword was used in all JavaScript code from 1995 to 2015.
- The let and const keywords were added to JavaScript in 2015.
- The var keyword should only be used in code written for older browsers.

### **Example using let:**

```
let x=25;  
let y=50;  
let z=x+y;  
console.log(z)
```

**Output:** PS D:\js rev\rajt3>node ./p1.js75

- in this above output we are getting outside of Browser by using Node.js

**Example: for run this t below program through browser**

```
<!DOCTYPEhtml>  
<html>  
<body>  
<h1>JavaScript Variables</h1>  
<p>In this example, x, y, and z are variables.</p>  
<pid="demo"></p>  
<script>  
let x = 25;  
let y = 50;  
let z = x + y;  
document.getElementById("demo").innerHTML="Sum of Two Number="+ z;  
</script>  
</body> </html>
```

### **Output:**

JavaScript Variables  
In this example, x, y, and z are variables.  
The value of z is: 75

### **Example:using const:**

```
const x = 25;  
const y = 50;  
const z = x + y;  
console.log(z)
```

**Output:** PS D:\js rev\rajt3>node ./p1.j75

### **Mixed Example:**

```
const Num1= 15  
const Num2 = 20
```

```
let Sum = Num1 + Num2  
console.log("Sum of Num1 and Num2=",Sum)
```

**Output:** Sum of Num1 and Num2= 35

**Note:**

- The two variables Num1 and Num2 are declared with the const keyword.
- These are constant values and cannot be changed.
- The variable Sum is declared with the let keyword.
- The value Sum can be changed.

**When to Use var, let, or const?**

1. Always declare variables
2. Always use const if the value should not be changed
3. Always use const if the type should not be changed (Arrays and Objects)
4. Only use let if you can't use const
5. Only use var if you MUST support old browsers.

**Example:**

```
<!DOCTYPEhtml>  
<html>  
<body>  
<h1>JavaScript Variables</h1>  
<p>In this example, x, y, and z are variables.</p>  
<form>  
<label>SUM</label>  
<input id="demo" type="text"></form>  
<p id="d1"></p>  
<script>  
let x = 25;  
let y = 50;  
let z = x + y;  
  
document.getElementById("demo").value = +z;  
document.getElementById("d1").innerHTML="sum of two numbers="+z;  
</script>  
</body></html>
```



# DIFFERENCE BETWEEN VAR CONST AND LET

	var	let	const
Hoisting	Hosted at top of global scope. Can be used before the declaration.	Hosted at top of some private scope and only available after assigning value. Can not be used before the declaration.	Hosted at top of some private scope and only available after assigning value. Can not be used before the declaration.
Scope	Global scope normally. Start to end of the function inside of the function.	Block scoped always. Start to end of the current scope anywhere.	Block scoped always. Start to end of the current scope anywhere.
Redeclaration	Yes, we can redeclare it in the same scope.	No, we can't redeclare it in the same scope.	No, we can't redeclare or <b>reinitialize</b> it.

Difference between var, let & const.

Scope → { ... code... }

## ❖ var:

- **Function-scoped:** Variables declared with var are function-scoped, meaning they are only accessible within the function in which they are declared. If declared outside of any function, they become globally scoped.
- **Hoisting:** Variables declared with var are hoisted to the top of their function or global scope. This means you can use a var variable before it's declared in code, but it will have an initial value of undefined.
- **Reassignable:** You can reassign values to a var variable.

## ❖ let:

- **Block-scoped:** Variables declared with let are block-scoped, meaning they are only accessible within the block (inside curly braces) in which they are declared, or within sub-blocks.
- **Hoisting:** Like var, let variables are hoisted, but they are not initialized to undefined. If you try to access a let variable before its declaration in code, you'll get a ReferenceError.
- **Reassignable:** You can reassign values to a let variable.

## ❖ const:

- **Block-scoped:** Like let, const variables are block-scoped.
- **Hoisting:** const variables are hoisted, but like let, they are also not initialized to undefined. Accessing a const variable before its declaration in code will result in a ReferenceError.
- **Immutability:** Variables declared with const cannot be reassigned once they are given a value. However, the value itself may be mutable if it's an object or an array. This means you can change the properties or elements of a const object or array.



**Example:**

- var is global scope
- let & const are block scope
- In JS, we can use of pair of empty curly brackets to create an empty block.

```
{  
var c=10  
let d=20  
    console.log(c,d)  
}  
console.log(c) //10  
console.log(d) //It will throw error as d can be accessed inside local block here  
conste=20 //const variables can't be declared separately but initialise also
```

**Example:**

```
let a="10"  
let b=20  
console.log(a+b++ +a) //a+b++ +a=102010  
console.log(a,b) //a="10" b=21  
console.log(a+b++ +b) //a+b+ ++b=102122
```

## ❖ **Block Scope:**

- Before ES6 (2015), JavaScript had Global Scope and Function Scope.
- ES6 introduced two important new JavaScript keywords: let and const.
- These two keywords provide Block Scope in JavaScript.
- Variables declared inside a {} block cannot be accessed from outside the block;

**Note:**

- let and const have block scope.
- let and const can not be redeclared.
- let and const must be declared before use.
- let and const does not bind to this.
- let and const are not hoisted.
- var does not have to be declared.
- var is hoisted.
- var binds to this.

## LET

- let Can not be Redeclared:
- Variables defined with let can not be redeclared.

**Example:**

```
let a=15  
console.log(a)  
let a=20  
console.log(a)
```

**Note:** But we can declare let in same name inside and outside of the {} block

**Example:**

```
let a=15  
{
```



```
let a=20
console.log("value of a=",a)
}
console.log("value of a=",a)
```

**Output:**

```
C:\Users\hp\OneDrive\Desktop\demo3> node jp.js
value of a= 20
value of a= 15
```

- let can be declare without assign value:

**Example:**

```
let fd // let can declear variable without assign value also
fd=50
console.log(fd)
```

**Output:** 50

## **VAR**

- var can be Redeclared.
- Variables defined with var can be redeclared allowed anywhere in a program.

**Example-1:**

```
var a=15
console.log("value of a=",a)
var a=20
console.log("value of a=",a)
```

**Output:**

```
value of a= 15
value of a= 20
```

**Example-2:**

```
Var a=15
console.log("Value of a out side of Block=",a)
{
Var a=20
console.log("value of a side of Block=",a)
}
console.log("value of a=",a)
```

**Output:**

```
Value of a out side of Block= 15
value of a side of Block= 20
value of a= 20
```

**Note:** var can be declare without assign value:

**Example:**

```
var fd // var can declear variable without assign value also
fd=25
console.log(fd)
```

**Output:** 25

# **CONST**

- The const keyword was introduced in ES6 (2015)
- Variables defined with const cannot be Redeclared
- Variables defined with const cannot be Reassigned
- Variables defined with const have Block Scope.

#### **Example-1:**

```
const a = 15
console.log("value of a=",a)
const a = 20 // Here x is 20 can not declare in the case of const
console.log("value of a=",a) // Here x is 15
```

#### **Example-2:**

- But const can be redeclared with same variable name inside and outside of {} block

#### **Example:**

```
const a = 15 // Here x is 15
{
  const a = 20 // Here x is 20
  console.log("value of a=",a)
}
console.log("value of a=",a) // Here x is 15
```

**Output:** value of a= 20  
value of a= 20

**Note:** const variable cannot be reassigned:

#### **Example:**

```
const fd= 30
fd = 9 // This will give an error
fd = fd + 23 // This will also give an error
console.log(fd)
• const Must be Assigned:
• JavaScript const variables must be assigned a value when they are declared:
```

#### **Example-1:**

```
constfd= 30
console.log(fd)
```

**Output:** 30

#### **Example-2:**

```
const fd // cantdeclare variable without assign value in case of const
fd=30 // error
console.log(fd)
```

## **❖ When to use JavaScript const?**

- Always declare a variable with const when you know that the value should not be changed.
- Use const when you declare:
- A new Array
- A new Object
- A new Function
- A new RegExp

# JAVASCRIPT OPERATORS

- Operators are symbolic representations that are used for performing various operations.
- JavaScript operators are symbols that are used to perform operations on operands.
- Types of JavaScript Operators
- There are different types of JavaScript operators:
  1. Arithmetic Operators
  2. Assignment Operators
  3. Comparison Operators
  4. Logical Operators
  5. Bitwise Operators
  6. Type Operators
  7. Ternary Operators

## ❖ Arithmetic Operators:

- Arithmetic Operators are used to perform arithmetic operations:

### Example:

Operator	Description	Example
➤ +	Addition	$15+20 = 35$
➤ -	Subtraction	$50-25 = 25$
➤ *	Multiplication	$3*6 = 18$
➤ /	Division	$40/8 = 5$
➤ %	Modulus (Remainder)	$20\%10 = 0$
➤ ++	Increment	<code>var a=20; a++; Now a = 21</code>
➤ --	Decrement	<code>var a=20; a--; Now a = 19</code>

## ❖ Assignment Operators:

- Assignment operators assign values to JavaScript variables.

### Example:

Operator	Description	Example
➤ =	Assign	$10+10 = 20$
➤ +=	Add and assign	<code>var a=10; a+=20; Now a = 30</code>
➤ -=	Subtract and assign	<code>var a=20; a-=10; Now a = 10</code>
➤ *=	Multiply and assign	<code>var a=10; a*=20; Now a = 200</code>
➤ /=	Divide and assign	<code>var a=10; a/=2; Now a = 5</code>
➤ %=	Modulus and assign	<code>var a=10; a%=2; Now a = 0</code>

## ❖ Comparison Operators or Relational :

Operator	Description	Example
➤ ==	Is equal to	$10==20 = \text{false}$
➤ ===	Identical (equal and of same type)	$10==20 = \text{false}$
➤ !=	Not equal to	$10!=20 = \text{true}$
➤ !==	Not Identical	$20!==20 = \text{false}$
➤ >	Greater than	$20>10 = \text{true}$
➤ >=	Greater than or equal to	$20>=10 = \text{true}$
➤ <	Less than	$20<10 = \text{false}$
➤ <=	Less than or equal to	$20<=10 = \text{false}$



## ❖ Logical Operators:

- The following operators are known as JavaScript logical operators.

Operator	Description	Example
> <b>&amp;&amp;</b>	Logical AND	(10==20 && 20==33) = false
> <b>  </b>	Logical OR	(10==20    20==33) = true
> <b>!</b>	Logical Not	!(10==20) = true

## ❖ Bitwise Operators:

- bitwise operators perform bitwise operations on operands. bitwise operators are as follows:

Operator	Description	Decimal	Example	Same as Result	
1. <b>&amp;</b>	AND	5 & 1	0101 & 0001	0001	1
2. <b> </b>	OR	5   1	0101   0001	0101	5
3. <b>~</b>	NOT	~5	~0101	1010	10
4. <b>^</b>	XOR	5 ^ 1	0101 ^ 0001	0100	4
5. <b>&lt;&lt;</b>	left shift	5 << 1	0101 << 1	1010	10
6. <b>&gt;&gt;</b>	right shift	5 >> 1	0101 >> 1	0010	2
7. <b>&gt;&gt;&gt;</b>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

## ❖ Type Operators:

Operator	Description
> <b>typeof</b>	Returns the type of a variable
> <b>instanceof</b>	Returns true if an object is an instance of an object type

## ❖ Ternary Operators:

- ternary operator, also known as the conditional operator, is a concise way to write conditional statements. It takes three operands and returns a value based on a condition.

### Syntax:

- > **condition ?expressionIfTrue : expressionIfFalse**
- **condition:** A boolean expression that is evaluated. If it's true, the expression before the : is executed; otherwise, the expression after the : is executed.
- **expressionIfTrue:** The value or expression to be returned if the condition is true.
- **expressionIfFalse:** The value or expression to be returned if the condition is false.

## ❖ Special Operators:

- The following operators are known as JavaScript special operators.

Operator	Description
> <b>,</b>	: Comma Operator allows multiple expressions to be evaluated as single statement.
> <b>delete</b>	: Delete Operator deletes a property from the object.
> <b>in</b>	: In Operator checks if object has the given property
> <b>new</b>	: creates an instance (object)
> <b>void</b>	: it discards the expression's return value.
> <b>yield</b>	: checks what is returned in a generator by the generator's iterator.



# **ARITHMETIC OPERATORS**

(+, -, \*, /, %, \*\*, ++, --)

## **Example:**

```
// let a=20
// let b=30
//DYNAMIC INPUT in node js
//Install the module prompt-sync
// npmi prompt-sync
const prompt = require("prompt-sync")();
let a = parseInt(prompt("Num 1- "))
let b = parseInt(prompt("Num 2- "))
let sum=a+b
let sub=a-b
let Mul=a*b
let Div=a/b
let modulus=a%b
let pre_inc=++a
let post_inc=a++
let pre_dec>--a
let post_dec=a--
console.log("Sum of a and b=",sum)
console.log("Subtraction of a and b=",sub)
console.log("Multiplication of a and b=",Mul)
console.log("Division of a and b=", Div.toFixed(3))
console.log("Modulus or Remainder of a and b=",modulus)
console.log("Pre increment value of a=",pre_inc)
console.log("Post increment value of a=",post_inc)
console.log("Pre decrement value of a=",pre_dec)
console.log("Post decrement value of a=",post_dec)
```

```
leta=100
letb=2
console.log(a**b)
console.log(100*100)
```

## **Output:**

```
PS D:\js rev\rajt3>npmi prompt-sync
added 3 packages in 850ms
PS D:\js rev\rajt3> node ./p1.js
Num 1- 20
Num 2- 10
Sum of a and b= 30
Subtraction of a and b= 10
Multiplication of a and b= 200
Division of a and b= 2
Modulus or Remainder of a and b= 0
Pre increment value of a= 21
Post increment value of a= 21
Pre decrement value of a= 21
Post decrement value of a= 21
```

## **Note:**

```
let a=20
console.log(a) //20
console.log(++a) //21
console.log(a++) //21
console.log(a)//22
console.log(--a)//21
console.log(a)//21
console.log(a--)//21
console.log(a)//20
```



## **ASSIGNMENT OPERATORS**

➤ = is assignment operator. It is used to store value inside a variable

**Example:**

```
a-=b //a=a-b  
console.log("value of a=", a)  
a*=b //a=a*b  
console.log("value of a=", a)  
a/=b //a=a/b  
console.log("value of a=", a)  
a%=b // a=a%b  
console.log("value of a=", a,b%a)  
b=2  
a**=b  
console.log("value of a=", a)
```

**OutPut:**

```
C:\Users\hp\OneDrive\Desktop\demo3>nodedemo3.js  
Enterthevalueofa=10  
Enterthevalueofb=20  
value of a=30  
value of a=10  
value of a=200  
value of a=10  
value of a=100  
value of a=100
```

## **COMPRESSION OPERATOR**

**Example:**

```
constprompt=require("prompt-sync")()  
leta=parseFloat(prompt("Enter the value of a"))// let a=20  
letb=parseFloat(prompt("Enter the value of b"))// let b=30  
console.log("comparing value of a and b is Equal",a==b)  
console.log("comparing value of a and b is Equal & same datatype",a==b)  
console.log("comparing value of a and b is Not Equal & same datatype",a!=b)  
console.log("comparing value of a and b is Not Equal",a!=b)  
console.log("comparing value of a and b, a is greater",a>b)  
console.log("comparing value of a and b, a is less than",a<b)  
console.log("comparing value of a and b , a is less than Equal",a<=b)  
console.log("comparing value of a and b, a is greater and Equal",a>=b)
```

**Output:**

```
C:\Users\hp\OneDrive\Desktop\demo3>nodedemo3.js  
Enterthevalueofa=10  
Enterthevalueofb=10  
comparingvalueofaandbisEqual true  
comparingvalueofaandbisEqual&samedatatype true  
comparingvalueofaandbisNotEqual&samedatatype false  
comparingvalueofaandbisNotEqual false  
comparingvalueofaandb, a is greater false  
comparingvalueofaandb, a is less than false  
comparingvalueofaandb , a is less than Equal true  
comparingvalueofaandb, a is greater and Equal true
```



# LOGICAL OPERATORS

- Logical operators are used to perform logical operations on Boolean values (True or False).

Operator	Description	Example
1) <b>&amp;&amp;</b>	Logical AND	$(10==20 \&\& 20==33) = \text{false}$
2) <b>  </b>	Logical OR	$(10==20 \mid\mid 20==33) = \text{false}$
3) <b>!</b>	Logical Not	$!(10==20) = \text{true}$

1. **and** → Returns True if both statements are true Ex:  $x < 5 \&\& x < 10$  ( True)
2. **or** → Returns True if one of the statements is true Ex:  $x < 5 \mid\mid x < 4$  (True)
3. **not** → Reverse the result, returns False if the result is true Ex:  $\text{not}(x < 5 \text{ and } x < 10)$

**AND Truth Table**

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

**OR Truth Table**

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

**NOT Truth Table**

A	B
0	1
1	0

**Note:** These all three operators allow to combine Boolean values or expressions to create more complex conditions.

## 1) For Boolean types behavior:

- True and False → false
- True or False → true
- not False → true

## 2) For non-Boolean types behavior:

- 0 means False
- Non-zero means True
- Empty string is always treated as False

**Example:**

```
const prompt=require("prompt-sync")()
let a=parseFloat(prompt("Enter the value of a="))
let b=parseFloat(prompt("Enter the value of b="))
// let a=20
// let b=30
let c=10
console.log(a>b&&a>c)
console.log(a>b | a>c)
console.log(!(a>b&&a>c))//(a>b && a>c) is false but adding NOT(!) makes the o/p true
console.log(!(a>b | a>c))//(a>b || a>c) is true but adding NOT(!) makes the o/p false
```

**Output:**

```
Enter the value of a=10
Enter the value of b=30
false
false
true
true
```



# BITWISE OPERATORS

- bitwise operators perform bitwise operations on operands. bitwise operators are as follows:

Operator	Description	Decimal	Example	Same as	Result
1) &	AND	5 & 1	0101 & 0001	0001	1
2)	OR	5   1	0101   0001	0101	5
3) ~	NOT	~ 5	~0101	1010	10
4) ^	XOR	5 ^ 1	0101 ^ 0001	0100	4
5) <<	left shift	5 << 1	0101 << 1	1010	10
6) >>	right shift	5 >> 1	0101 >> 1	0010	2
7) >>>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

➤ Bitwise operators are used to perform operations at the bit level on integers.

➤ These operators are applicable only for int and Boolean types.

➤ By mistake if we are trying to apply for any other type then we will get Error.

1) & → if both bits are 1 then only result is 1 otherwise result is 0.

2) | → if at least one bit is 1 then result is 1 otherwise result is 0.

3) ^ → if bits are different then only result is 1 otherwise result is 0.

4) ~ → bitwise complement operator    1 → 0 & 0 → 1

5) << → Bitwise left shift

6) >> → Bitwise right shift

## BITWISE OPERATOR TRUTH TABLES

AND "&"			OR " "		
INPUT 1	INPUT 2	OUTPUT	INPUT 1	INPUT 2	OUTPUT
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	1	1	1

XOR "^"			NOT "~"		
INPUT 1	INPUT 2	OUTPUT	INPUT	OUTPUT	
0	0	0	0	1	
0	1	1	1	0	
1	0	1			
1	1	0			

### Example-1



## ❖ Console.log(~5) → -6

### ❖ Explanation:

- The binary representation of 5 is 0101.
- after apply the bitwise NOT operator you get 1010.
- JavaScript uses two's complement representation for signed integers.
- note: (signed integers are a way to represent both positive and negative whole numbers. They include zero, all the positive whole numbers, and all the negative whole numbers.)
- range -2,147,483,648 to 2,147,483,647.)
- In two's complement, inverting the bits also involves negating the number. So, when you invert 1010, you get -1010 in binary.
- convert this binary representation to decimal, you can use the two's complement method:
  - Invert all the bits: -1010 becomes -0101.
  - Add 1 to the inverted number:  $-0101 + 1 = -0110$ .
- So, the final result is 0110, which is 6 in decimal.
- Therefore, the code print(~a) will output -6

- In digital computing, one's complement and two's complement are two different ways to represent signed integers (positive and negative numbers) using binary numbers.

### ❖ One's Complement:

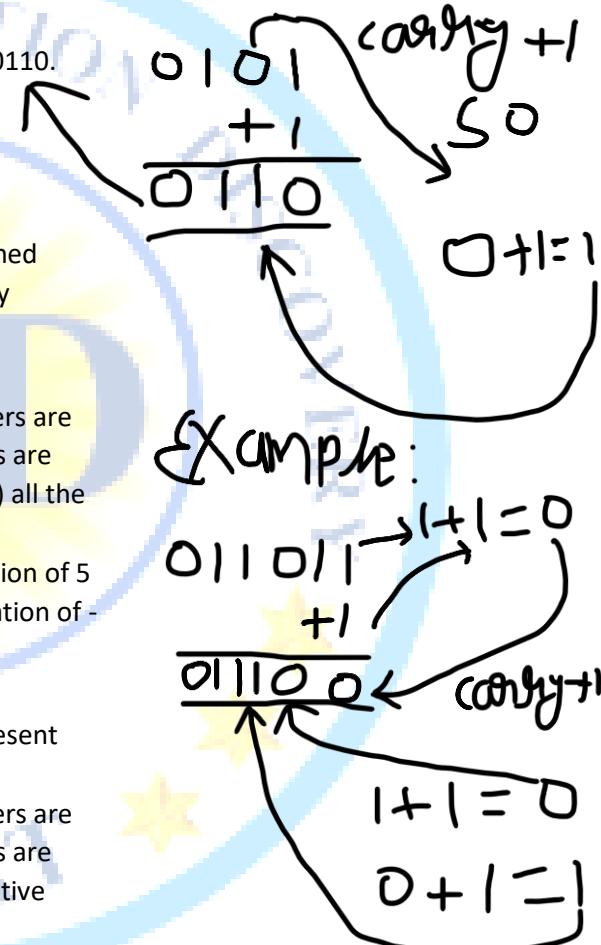
- In one's complement representation, positive numbers are represented as usual in binary, but negative numbers are obtained by inverting (changing 0s to 1s and 1s to 0s) all the bits of the corresponding positive number.
- For example, if you have the 8-bit binary representation of 5 as 00000101, then the one's complement representation of -5 would be 11111010.

### ❖ Two's Complement:

- Two's complement is the most common way to represent signed integers in modern digital computers.
- In two's complement representation, positive numbers are represented as usual in binary, but negative numbers are obtained by taking the one's complement of the positive number and then adding 1 to the result.
- For example, if you have the 8-bit binary representation of 5 as 00000101, then the two's complement representation of -5 would be 11111011.

### Note:

- If You want to add 1 to this binary number. When adding binary numbers, you start from the rightmost bit (the least significant bit) and move towards the left, just like when you add decimal numbers.
- The rightmost bits 1 + 1 equals 0, with a carry of 1.

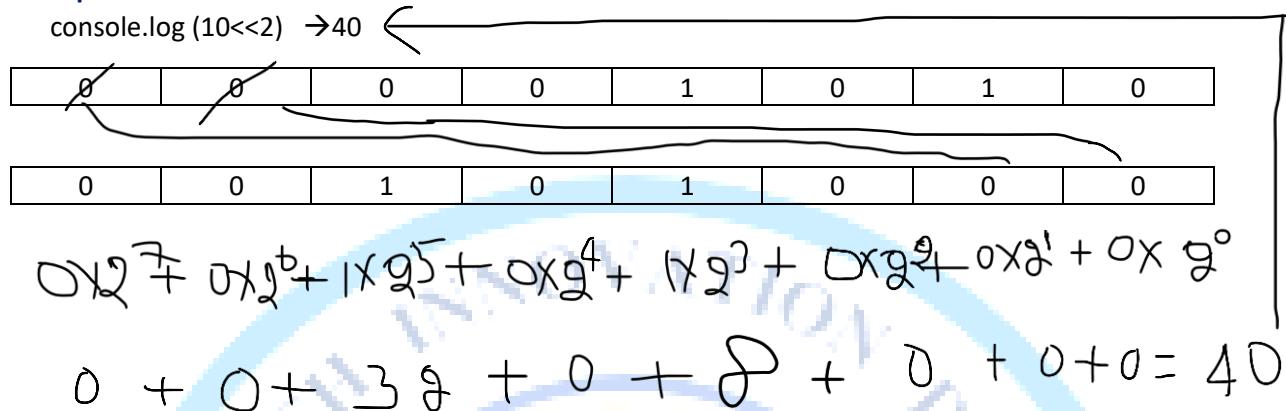


## SHIFT OPERATOR

- << Left shift Operator:

- After shifting the empty cells, we have to fill with zero.

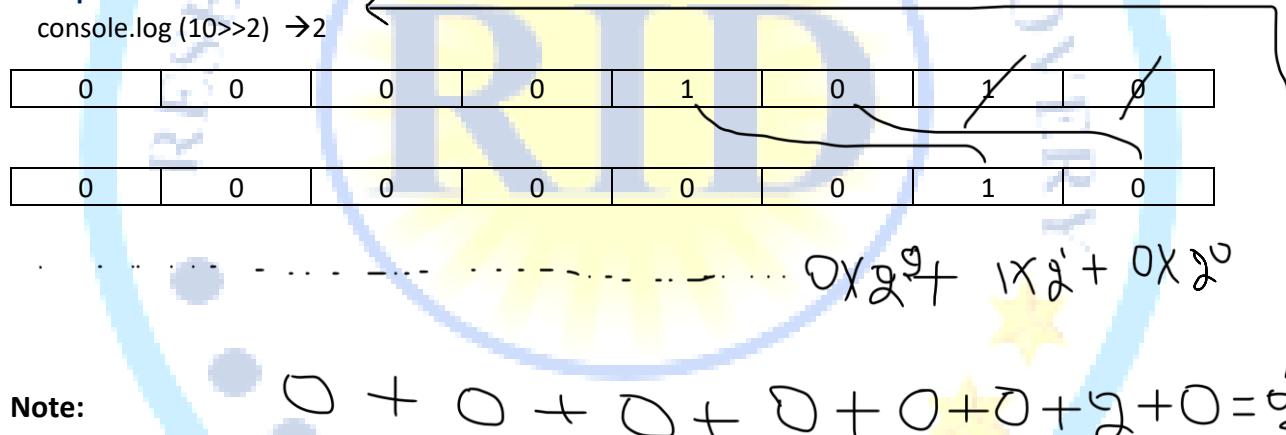
**Example:**



- >> Right shift Operator:

- After shifting the empty cells, we have to fill with sign bit. (0 for +Ve and 1 for -Ve)

**Example:**



**Note:**

- We can apply bitwise operators for Boolean types also

```
console.log (true&true)    > 1
console.log (true&false)   > 0
console.log (false&false)   > 0
console.log (false&true)    > 0
console.log (true|true)     > 1
console.log (true|false)    > 1
console.log (false|false)   > 0
console.log (false|true)    > 1
console.log (true^true)     > 0
console.log (true^false)    > 1
console.log (false^false)   > 0
console.log (false^true)    > 1
console.log(~true)          > -2
console.log(~false)         > -1
console.log(true<<2)       > 4
console.log(true>>2)       > 0
console.log(true>>>2)      > 0
```



# **TYPE OPERATORS**

➤ it is used for check data types of any variable

➤ **Operator              Description**

1. **typeof**              Returns the type of a variable

2. **instanceof**              Returns true if an object is an instance of an object type

**Example:**

```
consta=25;  
constcenter="T3 SKILLS CENTER";  
constb=true;  
let c  
constarry_c= [1, 2, 3, 4, 5, 6];  
constobj_data= { name:"Sangam", age:12 };  
constfun_2=function () {};  
//console.log("type of a=",typeof(a))  
console.log("Type of a=",typeof(a));  
console.log("Type of center=",typeof(center));  
console.log("Type of b=",typeof(b)); //  
console.log("Type of arry_c=",typeof(arra));  
console.log("Type of obj_data=",typeof(obj_data));  
console.log("Type of a=",typeof(fun_2));  
console.log(typeof(c));  
console.log(typeof(null)); // "object"  
(null is a special case)
```

**Output:**

Type of a= number  
Type of center= string  
Type of b= boolean  
Type of arry\_c= object  
Type of obj\_data= object  
Type of a= function  
undefined  
object

## ❖ **instanceof operator in JavaScript**

**Example:**

```
// Define constructor functions for custom objects  
functionAnimal(name) {  
    this.name=name;  
}  
functionDog(name, breed) {  
    Animal.call(this, name);  
    this.breed=breed;  
}  
constdog1=newDog("Buddy", "Golden Retriever");  
constdog2=newDog("Rex", "German Shepherd");  
constanimal=newAnimal("Unknown");  
// Use the instanceof operator to check object types  
console.log(dog1 instanceof Dog); // true, dog1 is an instance of Dog  
console.log(dog1 instanceof Animal); // true, dog1 is also an instance of Animal (due  
to prototype chain)  
console.log(animal instanceof Dog);  
console.log(animal instanceof Animal); // true, animal is an instance of Animal
```

**Output:**  
true  
false  
false  
true



## **TERNARY OPERATORS**

- **Operator:** The “Question mark” or “conditional” operator in JavaScript is a ternary operator that has three operands. It is the simplified operator of if/else.

**Syntax:**

- **condition ? "statements if condition is true" : "statements if condition is false"**

**Example:**

```
let n1=50  
let n2=20  
// if(n1>n2)  
// console.log(n1)  
// else  
// console.log(n2)  
let max=n1>n2?n1:n2  
console.log(max)
```

**Output:**50



# **DATA TYPES**

- Data types is type of value
- JavaScript provides different data types to hold different types of values.
- There are two types of data types in JavaScript.
  1. Primitive data type
  2. Non-primitive data type

**Note:** JavaScript is a dynamic type language, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine.

## **❖ Primitive data types**

- this is also known as inbuilt data types:
- There are following types of primitive data types
  - 1) Number
  - 2) String
  - 3) Boolean
  - 4) Undefined
  - 5) Null
  - 6) Symbol (ES6)
  - 7) BigInt (ES11)

### **❖ Number:**

- The Number data type is used to represent numeric values, which can be integers or floating-point numbers. Numbers are used for mathematical calculations and numerical operations.

**Example:**const age = 30; let b=3.6;

### **❖ String:**

- String data type is used to represent sequences of characters enclosed in single or double quotes.
- It is widely used for representing text and can include letters, numbers, symbols, and spaces.

**Example:**const name = "Sangam Kumar";

### **❖ Boolean:**

- The Boolean data type represents a binary value, which can be either true or false.
- Booleans are primarily used for logical operations and comparisons.

**Example:**const isStudent = true;

### **❖ Undefined:**

- Undefined data type represents a variable that has been declared but has not been assigned a value.
- Variables that are declared but not initialized are automatically assigned the value undefined.
- It is also used as the default return value of functions that do not explicitly return a value.

**Example:**let x; // x is undefined

### **❖ Null:**

- The Null data type represents the intentional absence of any object value or a placeholder for an object that does not exist or is unknown.
- It is often used to indicate that a variable intentionally has no value.

**Example:**const emptyValue = null;



### ❖ **Symbol (ES6):**

- Symbol data type, introduced in ECMAScript 6 (ES6), represents a unique and immutable value.
- Symbols are often used as object property keys to prevent unintentional property name collisions.
- They are guaranteed to be unique, even if two symbols have the same description.

**Example:** const uniqueKey = Symbol("unique");

### ❖ **BigInt (ES11):**

- The BigInt data type, introduced in ECMAScript 11 (ES11), is used to represent arbitrarily large integers that exceed the limits of the Number type.
- BigInts are created by appending n to the end of an integer literal or by using the BigInt() constructor.
- They are useful for working with very large numbers, such as when dealing with cryptographic operations or large integers.

**Example:** const bigintValue = 689012345678901234567890n;

#### **Program:**

```
const age=15;
const name="Sangam Kumar";
const isStudent=true;
let a;
Const emptyValue=null;
Const uniqueKey=Symbol("unique");
Const bigintValue=68901234567890n;
console.log(`Data type of age: ${typeof age}`);
console.log(`Data type of name: ${typeof name}`);
console.log(`Data type of isStudent: ${typeof isStudent}`);
console.log(`Data type of x: ${typeof x}`);
console.log(`Data type of emptyValue: ${typeof emptyValue}`);
console.log(`Data type of uniqueKey: ${typeof uniqueKey}`);
console.log(`Data type of bigintValue: ${typeof bigintValue}`);
```

#### **Output:**

```
Data type of age: number
Data type of name: string
Data type of isStudent: boolean
Data type of x: undefined
Data type of emptyValue: object
Data type of uniqueKey: symbol
Data type of bigintValue: bigint
```



# **NON-PRIMITIVE DATA TYPES**

- The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.
- Non-primitive data types in JavaScript, also known as reference types, are data structures that can hold multiple values and are mutable, including objects, arrays, and functions.

## **❖ Types of non-primitive data types:**

- 1) Object
- 2) Array
- 3) Function
- 4) Date
- 5) RegExp
- 6) Map
- 7) Set
- 8) WeakMap
- 9) WeakSet
- 10) Promise
- 11) Error
- 12) Symbol

- 1. Object:** Objects are complex data types that can store **key-value pairs** and represent various entities in JavaScript. They include regular objects, arrays, functions, and more.

### **Example:**

```
let student={  
    name:"Sangam",  
    Branch:"CSE",  
    Roll_N0:53,  
    Marks:90  
}  
console.log(typeof(student)) // object  
console.log(student.name) //sangam
```

- 2. Array:** Arrays are ordered lists of values, and they can contain elements of various data types. Arrays are used for storing collections of data.

### **Example:**

```
let colors= ["red", "green", "blue"];  
let data=[ "39", "Skills", "3.4", {name:"Raj"}]  
console.log(colors[0]); // "red"  
console.log(data[3])  
console.log(typeof(data))
```

- 3. Function:** Functions are reusable blocks of code that can be invoked to perform a specific task or calculation. Functions are also objects in JavaScript.

### **Example-1:**

```
function greet(name) {  
    return "Hello, "+name+"!";  
}  
console.log(greet("Sangam Kumar"));
```

### **Example-2:**

```
function greet(name){  
    console.log("hello every one")  
    return "Hello "+name  
}  
// console.log(typeof(greet()))  
console.log(greet("raj"))
```



**4. Date:** it represents dates and times. It is used for working with date and time-related operations.

**Example:** let today=new Date();

```
console.log(today); //current date and time  
console.log(typeof(today))
```

**5. RegExp:** The RegExp (regular expression) object is used for pattern matching within strings. It allows you to perform powerful text searching and manipulation.

**Example:**

```
let pattern= /world/;  
let text="Hello, world!";  
console.log(pattern.test(text)); // true
```

**6. Map:** The Map object is a collection of key-value pairs where keys can be of any data type. It is often used when you need to associate values with specific keys.

**Example:** let map=new Map();

```
map.set("name", "Sangam");  
map.set("age", 25);  
console.log(map.get("name")); // "Sangam"
```

**Note:** new keyword in JavaScript is used to create an **instance of an object** that has a constructor function or a class. It is crucial for creating objects based on templates, such as user-defined constructors or built-in JavaScript objects.

**7. Set:** it is a collection of unique values. It is used when you need to store a list of distinct items.

**Example:** let set=new Set();

```
set.add(1);  
set.add(2);  
set.add(2); // Duplicate value, will not be added  
console.log(set.size); // 2  
console.log(set)
```

## 8. WeakMap

- A WeakMap holds key-value pairs where the keys are objects, and the values can be arbitrary data. The keys are weakly referenced, allowing garbage collection.

**Example:**

```
let obj = { id: 1 };  
let weakMap = new WeakMap();  
weakMap.set(obj, "This is a weak reference");  
console.log(weakMap.get(obj)); // Output: "This is a weak reference"  
// If `obj` is deleted, it will be garbage-collected.
```

## 9. WeakSet

- It is a collection of objects, with each object being weakly referenced, allowing garbage collection.

**Example:**

```
let obj1 = { name: "sangam" };  
let obj2 = { name: "Mohan" };  
let weakSet = new WeakSet();  
weakSet.add(obj1);  
weakSet.add(obj2);  
console.log(weakSet.has(obj1)); // Output: true  
// If `obj1` or `obj2` is deleted, they will be garbage-collected.
```



## 10. Promise

- A Promise represents an asynchronous operation that can either succeed or fail.

```
let myPromise = new Promise((resolve, reject) => {
    let success = true; // Change this to false to simulate rejection
    if (success) {
        resolve("Promise resolved successfully!");
    } else {
        reject("Promise rejected.");
    }
});
myPromise
    .then((message) => console.log(message)) // For resolve
    .catch((error) => console.error(error)); // For reject
```

## 11. Error

 It represents an error in JavaScript, which can be thrown or caught in try-catch blocks.

### Example:

```
try {
    throw new Error("This is a custom error!");
} catch (error) {
    console.log(error.message); // Output: "This is a custom error!"
    console.log(error.name); // Output: "Error"
}
```

## 12. Symbol:

 Symbols can also be considered non-primitive, although they are often categorized as primitive data types. They are used for creating unique property keys in objects.

### Example:

```
let id=Symbol("id");
let user= {
    [id]:12345,
    name:"Sangam"
};
console.log(user[id]); // 12345
```

## Program:

### Object

```
const person = {
    firstName: "Sangam",
    lastName: "Kumar",
    age: 15
}
```

### Array

```
const numbers = [15, 20, 25, 50, 53]
// Function
function add(a, b) {
    return a + b
}
```

### Date

 const currentDate = new Date();

### RegExp

```
const regexPattern = /hello/i;
const sampleString = "TWKSAA SKILLS CENTER";
const isMatch = regexPattern.test(sampleString);
```



**Map**

```
constfruitMap = new Map();
fruitMap.set("apple", 3);
fruitMap.set("banana", 6);
fruitMap.set("cherry", 9);
Set constuniqueNumbers = new Set([1, 2, 3, 2, 4, 5, 4]);
```

**Symbol**

```
constuniqueKey = Symbol("unique");
console.log("Object:", person);
console.log("Array:", numbers);
console.log("Function Result:", add(3, 9));
console.log("Current Date:", currentDate);
console.log("RegEx Test Result:", isMatch);
console.log("Map:", fruitMap);
console.log("Set:", uniqueNumbers);
console.log("Symbol:", uniqueKey);
```

**Output:**

```
Object: { firstName: 'Sangam', lastName: 'Kumar', age: 15 }
Array: [ 15, 20, 25, 50, 53 ]
Function Result: 12
Current Date: 2023-09-14T14:25:32.365Z
RegEx Test Result: false
Map: Map(3) { 'apple' => 3, 'banana' => 6, 'cherry' => 9 }
Set: Set(5) { 1, 2, 3, 4, 5 }
Symbol: Symbol(unique)
```

**Example:** Check the data types:

```
Object const person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 15
};
```

```
Array const numbers = [1, 2, 3, 4, 5, 6];
```

**Function**

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

```
Date constcurrentDate = new Date();
```

**RegExp**

```
constregexPattern = /hello/i;
constsampleString = "TWKSAA SKILLS CENTER";
constisMatch = regexPattern.test(sampleString);
```

```
Map constfruitMap = new Map();
fruitMap.set("apple", 3);
fruitMap.set("banana", 6);
fruitMap.set("cherry", 9);
```

```
Set constuniqueNumbers = new Set([1, 2, 3, 2, 4, 5, 4]);
```

```
Symbol constuniqueKey = Symbol("unique");
```

#### Check data types

```
console.log(`Data type of person: ${typeof person}`); // "object"
console.log(`Data type of numbers: ${typeof numbers}`); // "object"
console.log(`Data type of add function: ${typeof add}`); // "function"
console.log(`Data type of currentDate: ${typeof currentDate}`); // "object"
console.log(`Data type of isMatch: ${typeof isMatch}`); // "boolean"
console.log(`Data type of fruitMap: ${typeof fruitMap}`); // "object"
console.log(`Data type of uniqueNumbers: ${typeof uniqueNumbers}`); // "object"
console.log(`Data type of uniqueKey: ${typeof uniqueKey}`); // "symbol"
```

#### Output:

```
Data type of person: object
Data type of numbers: object
Data type of add function: function
Data type of currentDate: object
Data type of isMatch: boolean
Data type of fruitMap: object
Data type of uniqueNumbers: object
Data type of uniqueKey: symbol
```

## ❖ Difference b/w primitive & non-primitive data types:

#### Example:

```
var bigNum =12342222222222222222222222222222n
```

**BigInt:** This data type can represent numbers greater than  $2^{53}-1$  which helps to perform operations on large numbers. The number is specified by writing 'n' at the end of the value  
console.log(typeof bigNum)

#### PRIMITIVE-

```
let num1=20;
let num2=num1;
console.log(num1,num2)
num1=30
console.log(num1,num2)
```

#### NON-PRIMITIVE

```
let a=[1,2,3]
let b=a
console.log(a,b)
a[1]=55
console.log(a,b)
```

**Note-** In case of primitive data type, the changes will be reflected in modified variable only

**But** In case of non-primitive data type, the changes will be reflected in both the copies

```
/*
Primitive data types hold immutable values
Non-Primitive data types hold mutable values.
*/
// let a=20;
// a=40;
// let c=[1,2,3]
// c[0]=20
```

#### Output:

```
bigint
20 20
30 20
[ 1, 2, 3 ] [ 1, 2, 3 ]
[ 1, 55, 3 ] [ 1, 55, 3 ]
```

# Base Conversion in JavaScript

- It is process of converting a number from one numerical base (or radix) to another.
- Base conversions often involve use of the `parseInt()` function and the `toString()` method.
- 1. **`parseInt(string, radix)`:** Parses a string and returns an integer of the specified radix (base).
- **Example:** `parseInt("2A", 16)` converts hexadecimal string "2A" to the decimal number 42.
- 2. **`Number.prototype.toString(radix)`:** Converts a number to a string specified radix (base).
- **Example:** `(42).toString(16)` converts the decimal number 42 to the hexadecimal string "2a".

## ❖ Convert to Decimal

1. **Binary to Decimal:** `parseInt(binaryString, 2)`
  - ✓ **Example:** `parseInt("101010", 2)` // returns 42
2. **Octal to Decimal:** `parseInt(octalString, 8)`
  - ✓ **Example:** `parseInt("52", 8)` // returns 42
3. **Hexadecimal to Decimal:** `parseInt(hexString, 16)`
  - ✓ **Example:** `parseInt("2A", 16)` // returns 42

## ❖ Convert from Decimal

1. **Decimal to Binary:** `decimalNumber.toString(2)`
  - ✓ **Example:** `(42).toString(2)` // returns "101010"
2. **Decimal to Octal:** `decimalNumber.toString(8)`
  - ✓ **Example:** `(42).toString(8)` // returns "52"
3. **Decimal to Hexadecimal:** `decimalNumber.toString(16)`
  - ✓ **Example:** `(42).toString(16)` // returns "2a"

## Convert Decimal to .....

### 1. Convert decimal to binary

**Example:**

```
let a=12  
let b=a.toString(2)  
console.log("Binary Number=",b)
```

**Output:**

Binary Number= 1100

### 2. Convert decimal to octal

**Example:**

```
let a=18  
let b=a.toString(8)  
console.log("Octal Number=",b)
```

**Output:**

Octal Number=22

### 3. Convert decimal to Hexadecimal

**Example:**

```
let a=139  
let b=a.toString(16)  
console.log("Hexadecimal Number=",b)
```

**Output:**

Binary Number= 8b

## Convert Binary to .....

### 1. Convert binary to decimal

**Example:**

```
let b = "101011";  
let d = parseInt(b, 2);  
console.log("Decimal=", d)
```

**Output:**

Decimal= 43

### 2. Convert binary to octal

**Example:**

```
let b = "101011";  
let d = parseInt(b, 2);  
let oct=d.toString(8)  
console.log("Octal Num=",oct)
```

**Output:**Octal Num=53

### 2. Convert binarytoHexadecimal

**Example:**

```
let b = "101011";  
let d = parseInt(b, 2);  
let Hex=d.toString(16)  
console.log("Octal Num=",Hex)
```

**Output:**Hex Num= 2b

## Convert Octal to .....

### 1.Convert octal to decimal

**Example:**

```
let octal_Num = "52";  
let d = parseInt(octal_Num, 8);  
console.log("Decimal_Num=", d);
```

**Output:**

Decimal\_Num= 42

### 2. Convert octal to binary

**Example:**

```
let Octal_Num = "52";  
let d = parseInt(Octal_Num, 8);  
let bin = d.toString(2);  
console.log("Binary_num=", bin);
```

**Output:**Binary\_num=101010

### 3. Convert octaltoHexadecimal

**Example:**

```
let Octal_Num = "52";  
let d = parseInt(Octal_Num, 8);  
let bin = d.toString(16);  
console.log("Binary_Num=", bin);
```

**Output:**Binary\_Num= 2a

## Convert Hexadecimal to .....

### 1.Convert Hexadecimal to decimal

**Example:**

```
let h = "2A";  
let d = parseInt(h, 16);  
console.log("Decimal Number=", d);
```

**Output:**

Decimal= 42

### 2. Convert Hexadecimal to octal

**Example:**

```
let h = "2A";  
let d = parseInt(h, 16);  
let oct=d.toString(8)  
console.log("Octal_no=", oct);
```

**Output:**Octal\_no=52

### 3. Convert Hexadecimal tobinary

**Example:**

```
let h = "2A";  
let d = parseInt(h, 16);  
let bin=d.toString(2)  
console.log("Binary Number=", bin);
```

**Output:**Binary Number= 101010



## RID Based Conversion

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>RID Base Conversion</title>
</head>
<body>
    <h1>RID Based Conversion Apps</h1>
    <label>Number</label>
    <input id="d1" type="text"><br><br>
    <label>From</label>
    <select id="from">
        <option value="d">Decimal Number</option>
        <option value="b">Binary Number</option>
        <option value="o">Octal Number</option>
        <option value="h">HexaDecimal Number</option>
    </select>
    <label>To</label>
    <select id="to">
        <option value="d">Decimal Number</option>
        <option value="b">Binary Number</option>
        <option value="o">Octal Number</option>
        <option value="h">HexaDecimal Number</option>
    </select>
    <button type="button" onclick="convert()">
        Convert
    </button>
    <br><br><br><br><br><br><br>
    <label>Result</label>
    <input id="res" type="text" >
<script>
//trim() it used for remove the space from left and
right side

```

```
function convert(){
    let num=document.getElementById("d1").value.trim()
    let frominp=document.getElementById("from").value
    let toinp=document.getElementById("to").value
    let res=""
    // convert user input into decimal // parseInt()
    let dec_num
    if(frominp==="b"){
        dec_num=parseInt(num, 2)
    }
    else if(frominp==="o"){
        dec_num=parseInt(num,8)
    }
    else if(frominp==="h"){
        dec_num=parseInt(num,16)
    }
    else if(frominp==="d"){
        dec_num=parseInt(num,10)
    }
    // convert decimal to any target number system
    // dec_num.toString()
    if(toinp==="d"){
        res= dec_num.toString(10)
    }
    else if(toinp==="b"){
        res=dec_num.toString(2)
    }
    else if(toinp==="o"){
        res=dec_num.toString(8)
    }
    else if(toinp==="h"){
        res=dec_num.toString(16)
    }
    document.getElementById("res").value=res
}
</script> </body> </html>
```

## RID Based Conversion Apps

Number

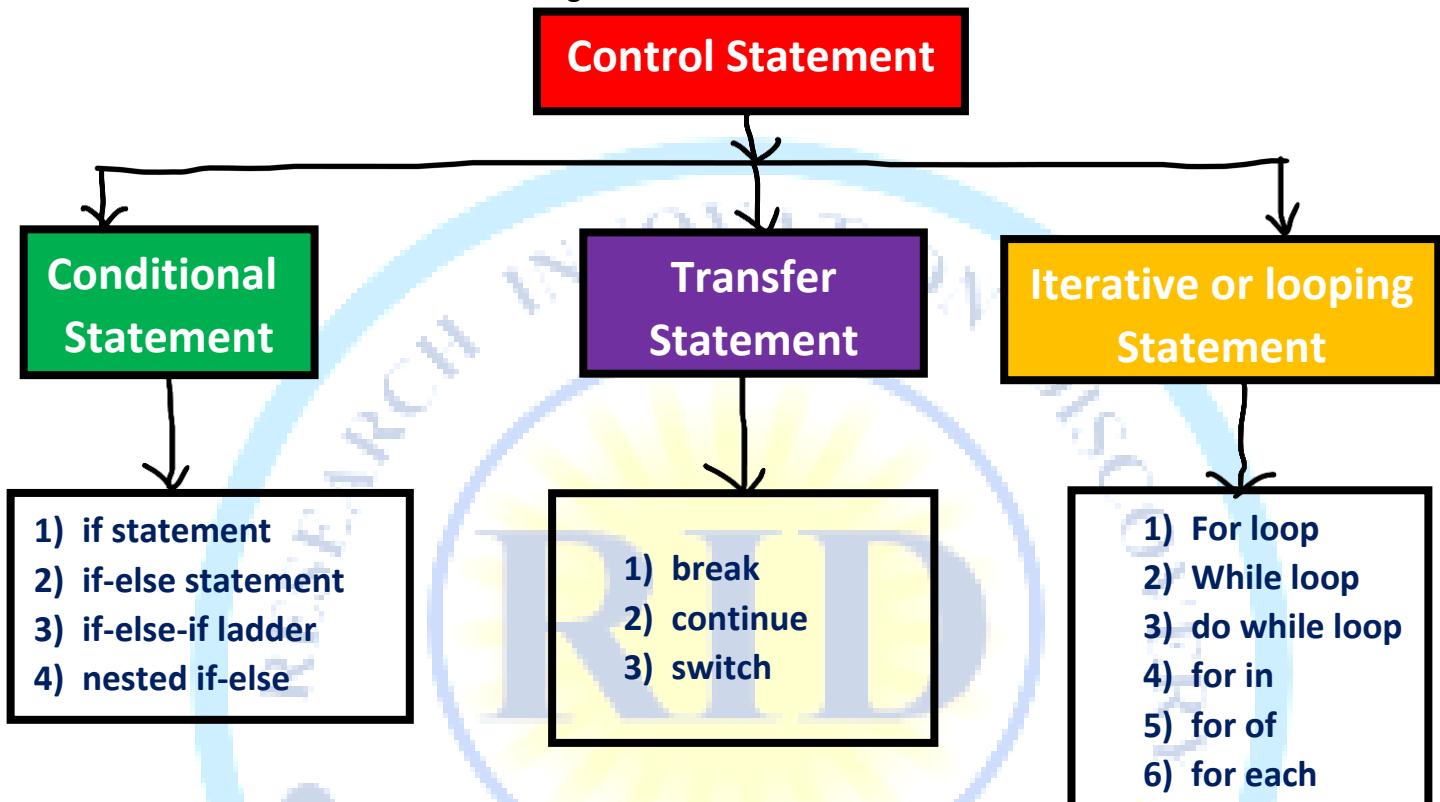
From  To  Convert

Result



# CONTROL STATEMENT

- Control Statement or Flow control describe the order in which statements will be executed at runtime.
- To change the programming follow based on the condition we will use control statement.
- This is used for Decision making.



- By using the {} we can create empty block in JavaScript

- Example:

```
{
```

```
Let a=10
```

```
Consol.log(a)
```

```
}
```

```
Consol.log(a) //can not access a outside of block
```

# CONDITIONAL STATEMENTS

- There are three main types of conditional statements:

1. if statements,
2. if-else statements,
3. else-if ladders.
4. Nested if-else

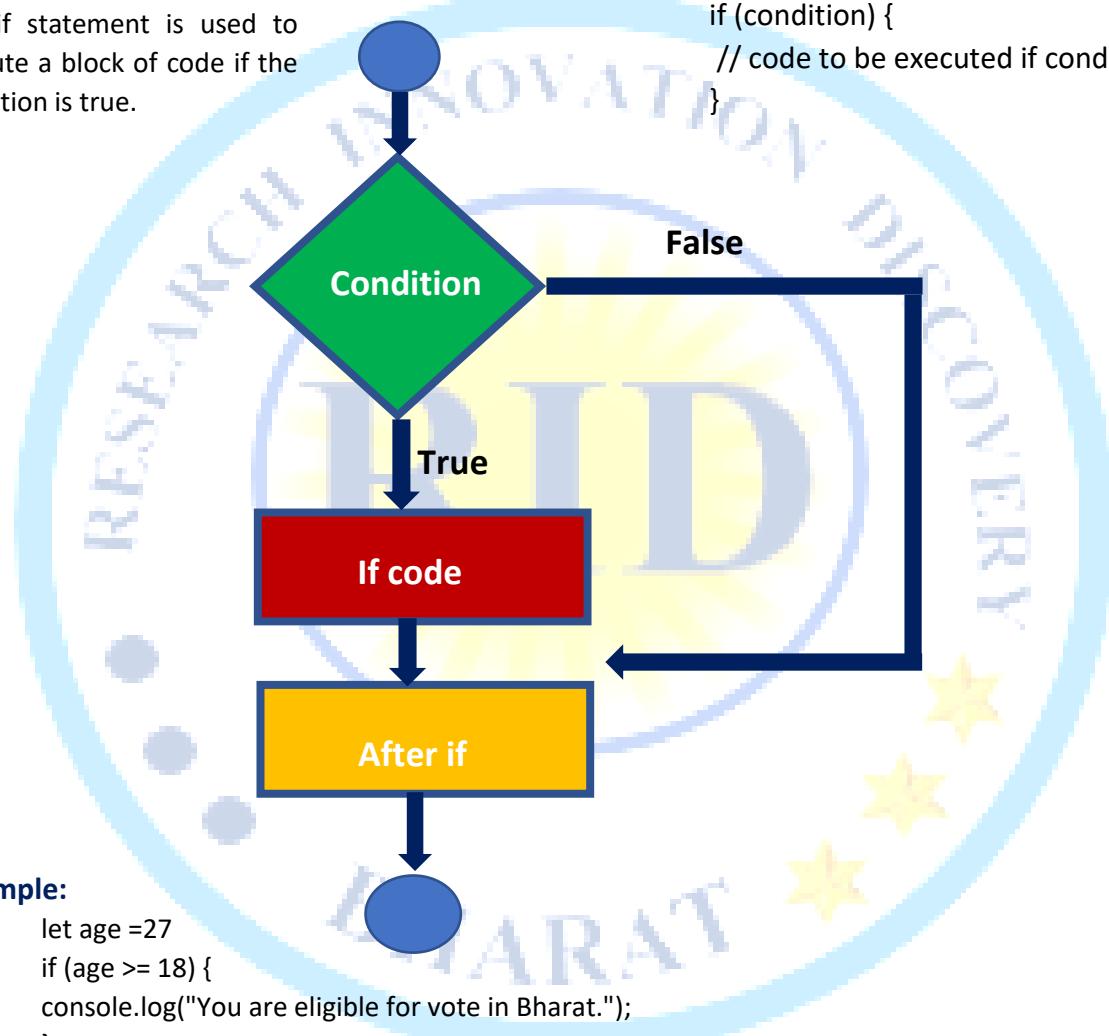
## 1) if statement:

- The if statement is used to execute a block of code if the condition is true.

### Flow Chart:

### Syntax:

```
if (condition) {  
    // code to be executed if condition is true  
}
```



### Example:

```
let age =27  
if (age >= 18) {  
    console.log("You are eligible for vote in Bharat.");  
}
```

**Output:** You are eligible for vote in Bharat.

- **Condition:** This is a Boolean expression that determines whether the code block inside the if statement should be executed.
- If the condition evaluates to true, the code block is executed; otherwise, it's skipped.

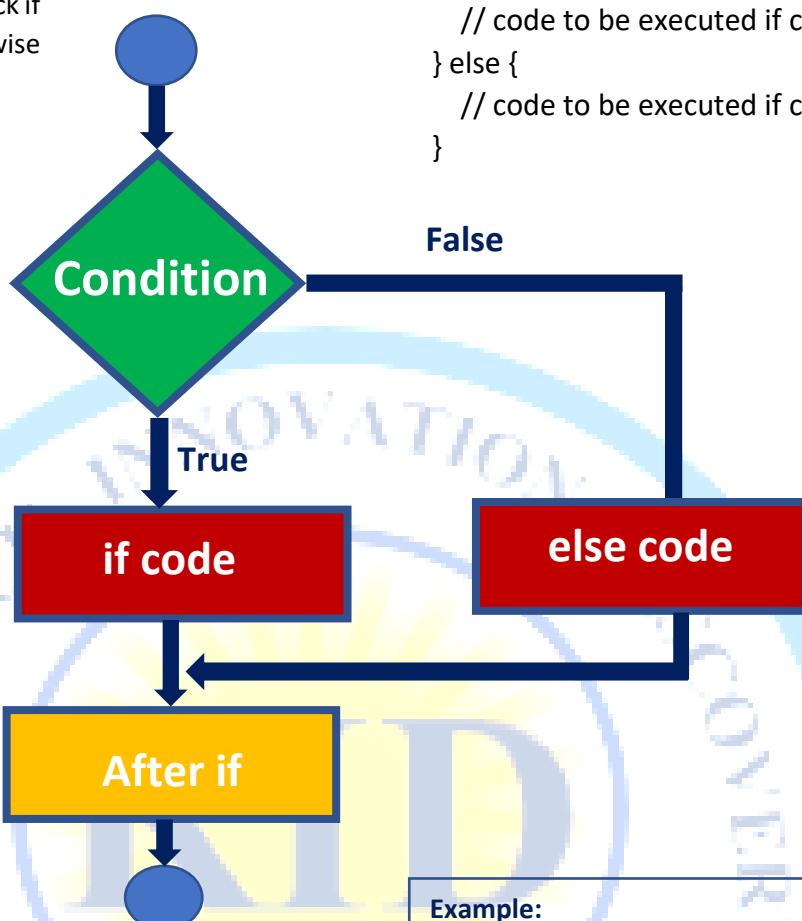
### Example-2:

```
let a=10  
If(a>0){console.log("+Ve")}  
If(a<0){console.log("-ve")}  
If(a==0{console.log("Zero")}
```

## 2) if else statement:

- if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

### Flow Chart:



### Syntax:

```

if (condition) {
    // code to be executed if condition is true
} else {
    // code to be executed if condition is false
}
  
```

#### Example-1:

```

let num = 15;
if (num > 0) {
    console.log("The number is positive.");
}
else {
    console.log("The number is not positive.");
}
  
```

**Output:** number is positive.

#### Example:

```

//DYNAMIC INPUT IN node js
//Install the module prompt-sync
// npm i prompt-sync
const prompt = require("prompt-sync")();
let num = parseInt(prompt("Enter the Number:"))
if (num > 0) {
    console.log("The number is positive.");
} else {
    console.log("The number is not positive.");
}
  
```

#### Example:

```

let num = 15;
if (num%2==0) {
    console.log("Even Number.");
}
else {
    console.log("Odd Number.");
}
  
```

#### Output:

odd number

#### Output:

Enter the Number:0  
The number is not positive.  
Enter the Number:-3  
The number is not positive.  
Enter the Number:15  
The number is positive.

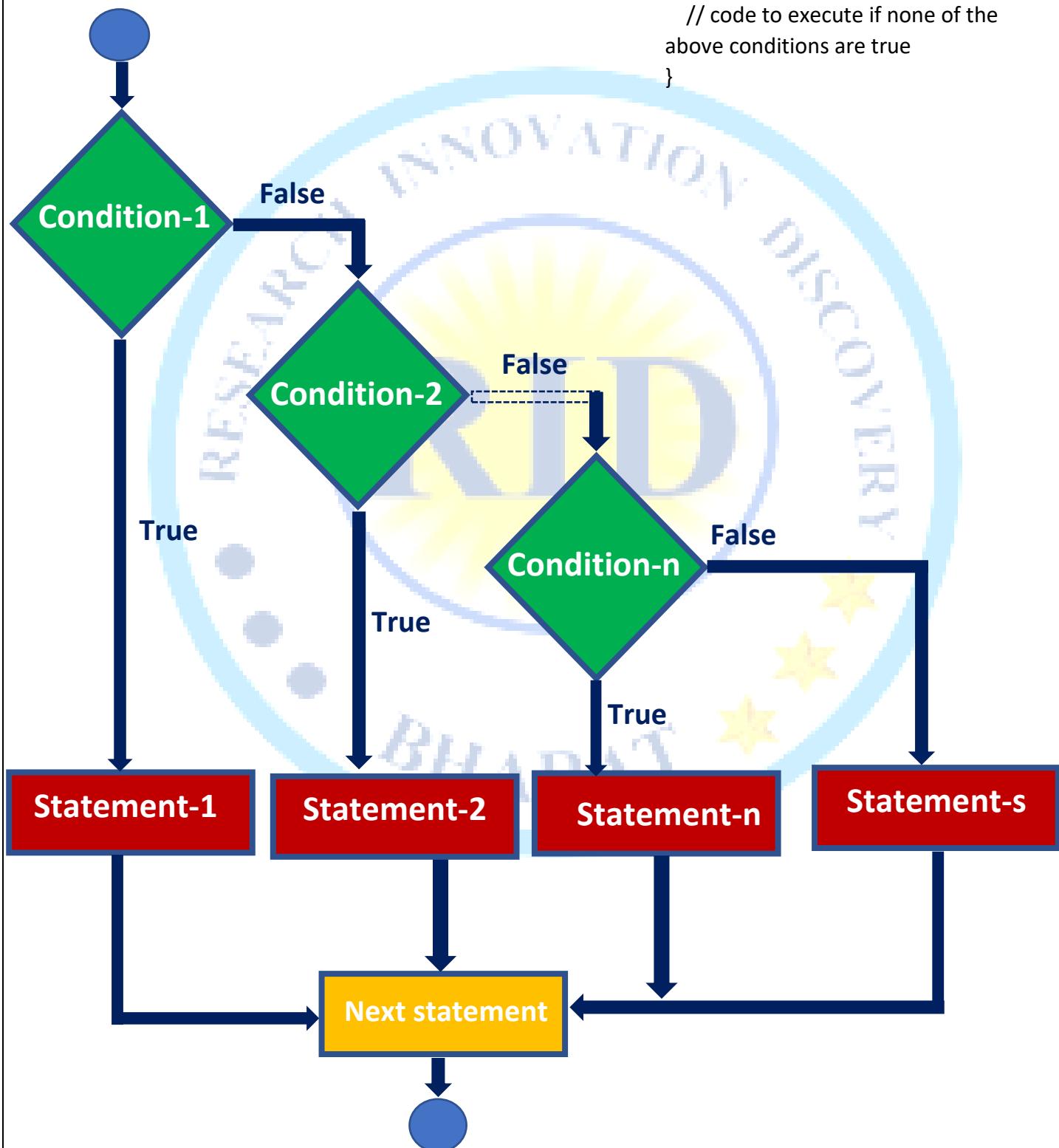
#### Question

- Check a number is positive or negative
- Check a number is odd or even
- Check a number is divisible by 5 or not
- Check a number is between 1 and 100 (inclusive)

### 3) if else if ladder statement:

- The if-else-if ladder statement is a series of if-else statements where each 'if' condition is checked sequentially until a true condition is found, or the final else block is executed if none of the conditions are true.

#### Flow Chart:



#### Syntax:

```

if (condition1) {
    // code to execute if condition1 is true
} else if (condition2) {
    // code to execute if condition2 is true
} else if (condition n) {
    // code to execute if condition3 is true
} else {
    // code to execute if none of the
    above conditions are true
}
  
```

### Example-1:

```
//DYNAMIC INPUT IN node.js
//Install the module prompt-sync
// npm i prompt-sync
const prompt = require("prompt-sync")();
let score = parseInt(prompt("Enter the Marks:"));
if (score >= 90) {
    console.log("You got an A Grade.");
}
else if (score >= 80) {
    console.log("You got a B Grad.");
}
else if (score >= 70) {
    console.log("You got a C Grad.");
}
else if (score >= 60) {
    console.log("You got a D Grad.");
}
else if (score >= 50) {
    console.log("You got a E Grad.");
}
else {
    console.log("Fail!!!!");
}
```

### Output:

Enter the Marks:91  
 You got an A Grade.  
 Enter the Marks:59  
 You got a E Grad.  
 Enter the Marks:45  
 Fail!!!

### Example-2:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
<style>
    #d1{
        text-align: right;
        font-size: 30px;
    }
</style>
</head>
<body>
    <label for="d1">Marks</label>
    <input id="d1" type="text" placeholder="Enter your marks"><br><br>
    <button type="button" onclick="grade()">Check Grade</button><br><br>
    <label for="d2">Grade</label>
    <input type="text" id="d2">
    <script>
        function grade() {
            let marks = parseFloat(document.getElementById("d1").value);
            let grade;
            if (marks >= 90 && marks <= 100) {
                grade = "A";
            } else if (marks >= 80 && marks < 90) {
                grade = "B";
            } else if (marks >= 70 && marks < 80) {
                grade = "C";
            } else if (marks >= 60 && marks < 70) {
                grade = "D";
            } else if (marks >= 50 && marks < 60) {
                grade = "E";
            } else {
                grade = "Fail";
            }
            document.getElementById("d2").value = grade;
            document.getElementById("d1").value = "";
        }
    </script>
</body>
</html>
```



### **Example: Check given character is lowercase uppercase, digit or special case**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<label for="d1">Character</label>
<input id="d1" type="text" placeholder="Enter your marks"><br><br>
<button type="button" onclick="check()">Check Grade</button><br><br>
<label for="d2">Grade</label>
<input type="text" id="d2">
<script>
function check() {
    let char = document.getElementById("d1").value;
    let res;
    if (char >= 'a' && char <= 'z') {
        res = "Lower case";
    } else if (char >= "A" && char <= "Z") {
        res = "Upper Case";
    } else if (char >= "0" && char <= "9") {
        res = "Digit";
    }
    else {
        res = "Special symbol";
    }
    document.getElementById("d2").value = res;
    document.getElementById("d1").value = " ";
}
</script>
</body>
</html>
```

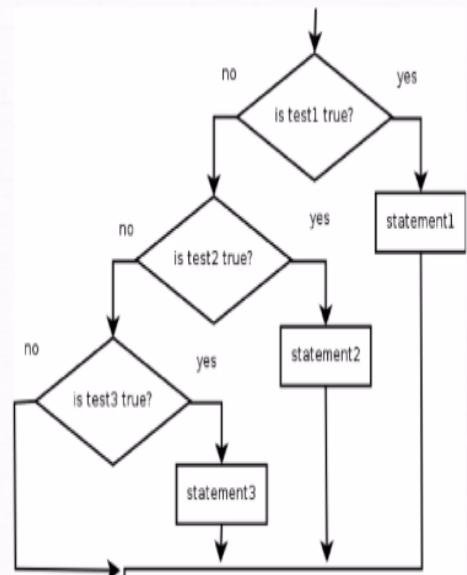
## 4. Nested If else:

- A nested if-else is an if-else structure inside another if-else block, allowing complex decision-making.

```
if(Condition/Expression) ← Outer If Block
{
    if(Condition/Expression) ← Inner If Block
    {
        Outer if and Inner If Statements;
    }
    else ← Inner Else Block
    {
        Outer if and inner else statements;
    }
}
else ← Outer Else Block
{
    if(Condition/Expression) ← Inner If Block
    {
        Outer else and inner if statements;
    }
    else ← Inner Else Block
    {
        Outer else and inner else statements;
    }
}
```

## Nested if else if Flow Chart

```
if (<test>) {
    <statement(s)>;
} else if (<test>) {
    <statement(s)>;
} else if (<test>) {
    <statement(s)>;
}
```



## 2. Login Successful Example

```
let username = "RID";
let password = "1234";
if (username === "RID") {
    if (password === "1234") {
        console.log("Login Successful!");
    } else {
        console.log("Incorrect Password!");
    }
} else {
    console.log("User not found!");
}
```

## 1. Grading System

```
let marks = 87;
if (marks >= 50) {
    if (marks >= 85) {
        console.log("Grade: A+");
    } else if (marks >= 70) {
        console.log("Grade: A");
    } else {
        console.log("Grade: B");
    }
} else {
    console.log("Failed.");
}
```

## 4. Online Store Purchase Decision

```
let balance = 100;
let itemPrice = 75;
let isInStock = true;

if (balance >= itemPrice) {
    if (isInStock) {
        console.log("Purchase successful!");
    } else {
        console.log("Item is out of stock.");
    }
} else {
    console.log("Insufficient balance.");
}
```

## 3. Age and License Eligibility

```
let age = 20;
let hasLearnerPermit = true;
if (age >= 18) {
    if (hasLearnerPermit) {
        console.log("You are eligible to apply for a driving license.");
    } else {
        console.log("You need a learner's permit first.");
    }
} else {
    console.log("You are not old enough to apply for a license.");
}
```



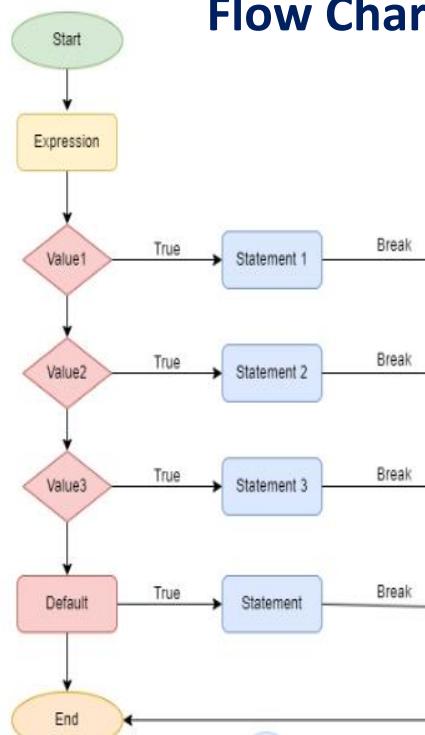
## ❖ Switch Statement:

- Use the switch statement to select one of many code blocks to be executed.

**Syntax:** switch(expression) {

```
case x:  
    // code block  
    break;  
case y:  
    // code block  
    break;  
default:  
    // code block}
```

## Flow Chart



## ❖ How it's works:

- The switch expression is evaluated once. The value of the expression is compared with the values of each case. If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

**Question:** Program to print the correct week name as per given week number

1-Monday, 2-Tuesday, 3-Wednesday, 4-Thursday, 5-Friday, 6-Saturday, 7-Sunday

**Program:**

```
const prompt = require("prompt-sync")();  
let week = parseInt(prompt("Enter Your Choice:"))  
switch(week){  
    case 1: console.log("Monday")  
        break  
    case 2: console.log("Tuesday")  
        break  
    case 3: console.log("Wednesday")  
        break  
    case 4: console.log("Thursday")  
        break  
    case 5: console.log("Friday")  
        break  
    case 6: console.log("Saturday")  
        break  
    case 7: console.log("Sunday")  
        break  
    default:  
        console.log("Wrong input")}
```

**Output:** Enter Your Choice:3

Wednesday

### Example: Calculator

```
const prompt = require("prompt-sync")();  
let num1 = parseFloat(prompt("Enter the first number: "));  
let num2 = parseFloat(prompt("Enter the second number: "));  
let operator = prompt("Enter an operator (+, -, *, /): ");  
switch (operator) {  
    case '+':  
        console.log(`Result: ${num1 + num2}`);  
        break;  
    case '-':  
        console.log(`Result: ${num1 - num2}`);  
        break;  
    case '*':  
        console.log(`Result: ${num1 * num2}`);  
        break;  
    case '/':  
        if (num2 !== 0) {  
            console.log(`Result: ${num1 / num2}`);  
        } else {  
            console.log("Error: Division by zero");  
        }  
        break;  
    default:  
        console.log("Invalid operator");}
```

**Output:** Enter the first number: 10  
Enter the second number: 20  
Enter an operator (+, -, \*, /): +  
Result: 30



**Problem:** write a program to print the grade according to given marks:

```
let n = 91;
switch (true) {
  case (n >= 90 && n <= 100):
    console.log("A");
    break;
  case (n >= 80 && n < 90):
    console.log("B");
    break;
  case (n >= 70 && n < 80):
    console.log("C");
    break;
  case (n >= 60 && n < 70):
    console.log("D");
    break;
  default:
    console.log("not correct");
}
```

## ❖ Break and Continue

- break statement in JavaScript is used to terminate a loop or switch statement prematurely.
- break statement in JavaScript exits a loop or switch block immediately, transferring control to the code following it.

### Example:

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    break; // Exit the loop when i equals 3
  }
  console.log(i);
}
```

**Output:** 1, 2

### ❖ Continue:

- continue statement in JavaScript skips the current iteration of the loop and proceeds with the next iteration.

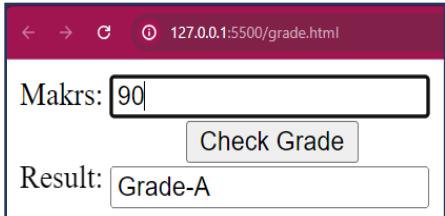
### Example:

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue; // Skip the rest of the code
    for this iteration
  }
  console.log(i);
}
```

**Output:** 1, 2, 4, 5

```
<!DOCTYPE html>
<html lang="en"><head>
<meta charset="UTF-8">
<title>Document</title>
</head><body>
<form onsubmit="sub(event)">
<label for="d1">Marks:</label>
<input type="text" placeholder="Enter Marks" id="d1">
<!--<button type="submit" onclick="check()">Check Grade</button><br><br>-->
<button type="submit">Check Grade</button><br><br>
</form>
<label for="d2">Result:</label>
<input type="text" id="d2">
<script>
  function sub(event){
    event.preventDefault()
    check()
  }

  function check() {
    let marks = parseFloat(document.getElementById("d1").value)
    document.getElementById("d1").value = ""
    let result =""
    if(!isNaN(marks)){
      switch (true) {
        case (marks >= 90 && marks <= 100):
          result = "Grade-A"
          break
        case (marks >= 80 && marks < 90):
          result = "Grade-B"
          break
        case (marks >= 70 && marks < 80):
          result = "Grade-C"
          break
        case (marks >= 60 && marks < 70):
          result = "Grade-D"
          break
        case (marks >= 50 && marks < 60):
          result = "Grade-E"
          break
        default:
          result = "Fail"
      }
    }
    else{
      result="Invalid Number"
    }
    document.getElementById("d2").value = result
  }
</script>
</body>
</html>
```




# LOOP STATEMENT

- Loop is also known as iterative method it is used for execute code repletely.
- Loop statement is specially used for repeat the same code util condition is true.
- There are two major categories of loop -
  1. **Entry controlled(for,while):-**The loop body is executed if condition is true.
  2. **Exit controlled(do-while):-**The loop body executes first & then condition is checked.

**NOTE**-if condition is false, then loop body is not executed in case of for/while but will execute in do-while once

## ❖ **For Loop:**

- it is executing a block of code repeatedly for a specified number of times or until a certain condition is met.

### Syntax:

```
for (initialization; condition; increment/decrement) {  
    // Code to be executed in each iteration  
}
```

### ➤ Syntax Explanation:

- **initialization:**This part is typically used to initialize a variable before the loop starts. It's executed only once, at the beginning.
- **condition:**This is the condition that is evaluated before each iteration. If the condition is true, the loop continues; if it's false, the loop stops.
- **increment/decrement:**After each iteration of the loop, this part is used to update the loop control variable. It can be an increment (e.g., i++ to increase by 1) or a decrement (e.g., i-- to decrease by 1).
- **Code to be executed in each iteration:**This is the block of code that is executed repeatedly as long as the condition remains true.

### Example:

```
for (let i = 1; i<= 6; i++) {  
    console.log(i);  
}
```

### Output:1

```
2  
3  
4  
5  
6
```

### Explanation:

let i = 1 initializes the loop control variable i to 1.

i<= 6 is the condition, and as long as i is less than or equal to 5, the loop continues.

i++ increments i by 1 after each iteration.

console.log(i) prints the value of i in each iteration.

**Q1.** Write a program to find sum of all even numbers from 1 to 100:

**Q2.** Write to find the sum of all natural from 1 to 10.

**Q3.** Write a program to print the 10 times of your name

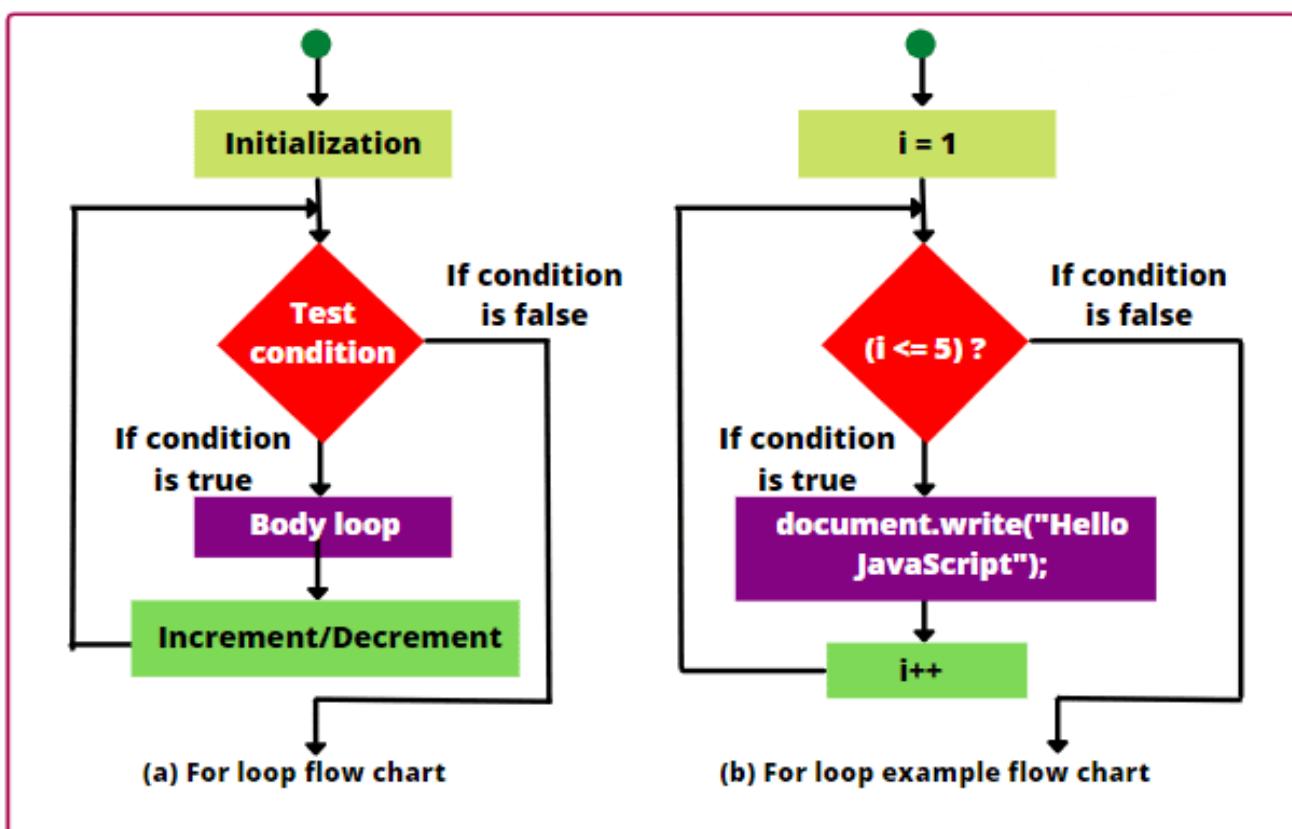
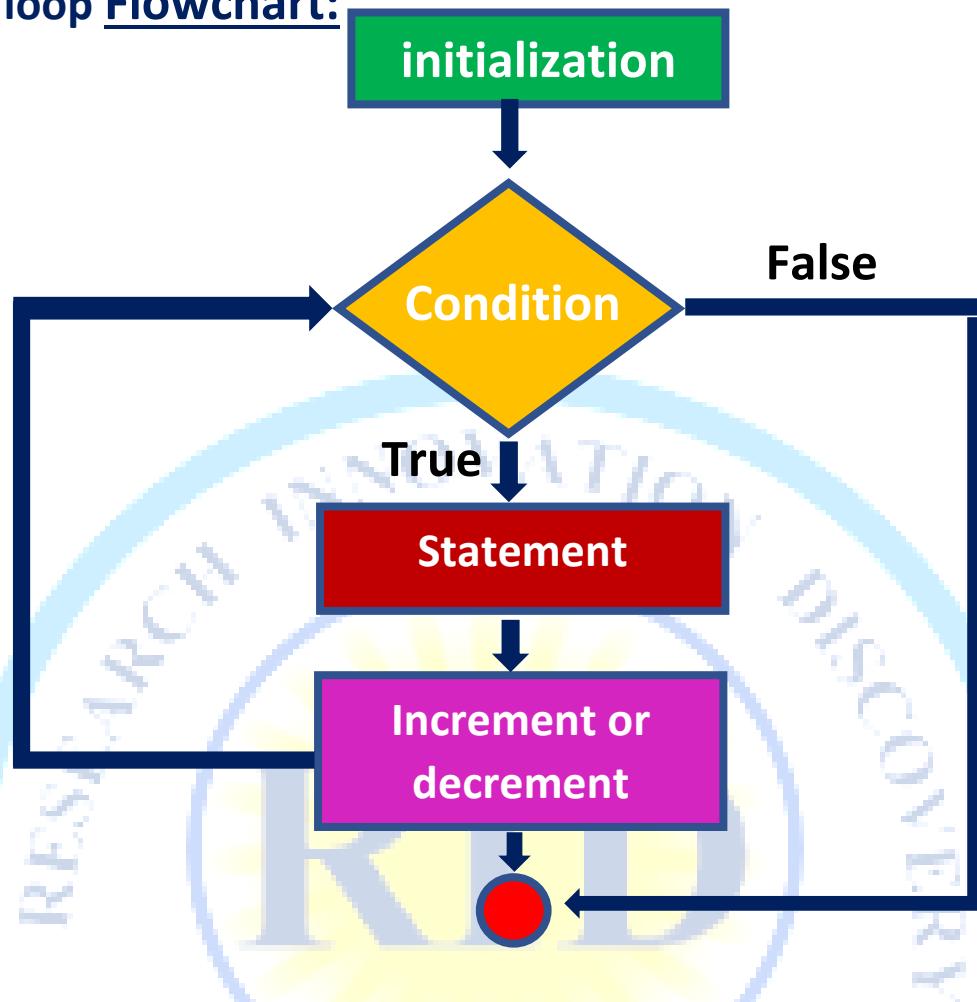
### #How to Print output in one line

```
let n=10  
let output=""  
for (let i=0; i<=n; i++){  
    output=output+i+" "; // output+=i+" "  
}  
console.log("Number=",output)
```

### Output:

```
Number= 0 1 2 3 4 5 6 7 8 9 10
```

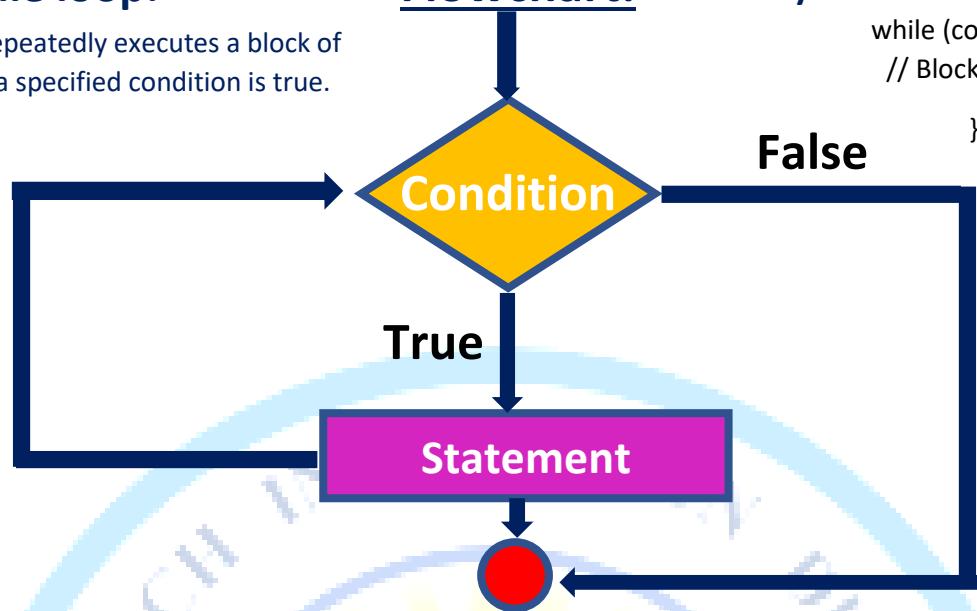
### ❖ For loop Flowchart:



## ❖ While loop:

The while loop repeatedly executes a block of code as long as a specified condition is true.

## Flowchart:



## Syntax:

```
while (condition) {
  // Block of code
}
```

**Entry**

**Entry**

**Test condition**

**If condition is false**

**count = 0;**

**If condition is false**

**Body of the loop**

**If condition is true**

**document.write("Hello world");  
count++;**

(a) While loop flow chart

(b) While loop example flow chart

### Example:

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

### Output:

```
1
2
3
4
```

```
let n = 10;
let i = 0;
let output = "";
while (i <= n) {
  output += i + " ";
  i++;
}
console.log("Number =", output);
output:
Number = 0 1 2 3 4 5 6 7 8 9 10
```

### Q.Find the sum of even numbers

```
let n=100
let i=0
let s=0
while(i<=n){
  if (i%2==0){
    s=s+i
  }
  i++
  // console.log("sum=",s)
}
console.log("sum=",s)
Output:
sum=2550
```

## ❖ do...while Loop:

- The do...while loop is similar to the while loop, but it ensures that the block of code is executed at least once, even if the condition is false initially.

### Syntax:

```
do {
    // Code to be executed at least once
} while (condition);
```

#### Example-1:

```
n=5
do{
    console.log("-ve Number ")
} while(n<0) // False then also one time block of code are execute
```

**Output:** -ve Number

Where as by using the while loop

**Ex:**

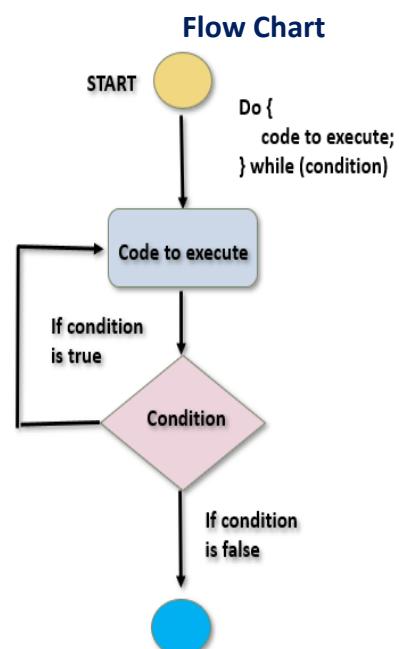
```
n=5
while(n<0){
    console.log("-ve")
}
```

#### Example-2:

```
let i=5;
do{
    console.log(i)
    i--
}while(i>=1)
do{
    console.log(i)
    i--
}while(i>6)
```

**Output:** 5

```
4
3
2
1
0
```



#### Example-3:

```
let n=10
let i=0
do{
    console.log(i)
    i++
}while(i<=n)
```

Or

```
letn=10
leti=0
output=" "
do{
    output+=i+" "
    i++
}while(i<=n)
console.log("Number=",output)
```

**Output:**

```
Number= 0 1 2 3 4 5 6 7 8 9 10
```

#### Q. Find the sum of even numbers

```
letn=100
lets=0
leti=1
do{
    if(i%2==0){
        s=s+i
    }
    i++
}while(i<=100)
console.log("sum=",s)
```

**Output:**

```
sum= 2550
```

## ❖ for in loop:

- `for...in` :- used for non-primitive data type
- `for...in` loop is used to iterate over properties of an object. It's often used for object iteration.

### Syntax:

```
for (const property in object) {
    // Code to be executed for each property
}
```

### Example:

```
let obj={name:"Sangam Kumar",age:15}
for(let k in obj){
    console.log(k,obj[k])}
```

**Output:** Name Sangam Kumar

age 15

### Example:

```
const person = {
    name: "Sangam Kumar",
    age: 15,
    job: "Developer";
    for (const key in person) {
        console.log(key + ": " + person[key]); }
```

**Output:** name: Sangam Kumar

age: 15

job: Developer

### Example:

```
let arr=[10,20,"Red",50.2]
for(let i in arr)
    console.log(i,arr[i])
for(let i in arr){
    if(i==2)
        console.log(arr[i])}
```

**Output:** 0 10

1 20

2 Red

3 50.2

### Example:

```
let obj={Name:"Sangam Kumar",age:15}
for(let i in obj){
    if(obj[i]==15)
        console.log(obj[i])}
```

**Output:** 15

## ❖ for...of Loop:

- `for...of` loop is used to iterate over elements of iterable objects like **arrays**, strings, etc.

**Note-if key & value both are required, use `for..in` otherwise if value is required, use `for..of`**

### Syntax:

```
for (const element of iterable) {
    // Code to be executed for each element
}
```

### Example-1:

```
let s="SKILLS"
for(let i of s)
    console.log(i)
```

**Output:** S

K

I

L

L

S

### Example-2:

```
let s="RID BHARAT"
for (let i of s){
    console.log(i)
}
```

**Output:** R I D B H A R A T

### Example-3:

```
let s=[10,20,30,40]
for (let i of s){
    console.log(i)
}
```

**Output:** 10 20 30 40

### Example-4:

```
let s=[10,20,30,40]
for (let i in s){
    console.log(i) // 0 1 2 3 this all are
    index value
    console.log(s[i]) // here i is a index
    value
}
```

**Output:** 10 20 30 40

### Example:

```
const skills = ["Python", "HTML", "CSS", "JavaScript", "Reactjs", "Node JS"];
for (const i of skills) {
    console.log(i);
}
```

**Output:** Python  
HTML  
CSS  
JavaScript  
React js  
Node JS



❖ forEach():

- The forEach() method calls a function for each element in an array.
- ForEach() method is a convenient way to iterate over the elements of an array or any iterable object and perform an operation or execute a function for each element.

**Syntax:**

```
array.forEach((callback,[index],[array] ){  
    // Code to be executed for each element  
})
```

- **array:** The array (or any iterable object) that you want to iterate over.
- **callback:** A function that gets executed for each element in the array.
- **currentValue:** The current element being processed in the array.
- **index (optional):** The index of the current element being processed.
- **array (optional):** The array on which forEach() was called.

**Example:**

```
Const colors = ['red', 'green', 'blue']  
colors.forEach(function(color, index) {  
    console.log(`Color at index ${index}: ${color}`)  
})
```

**Output:**

```
Color at index 0: red  
Color at index 1: green  
Color at index 2: blue
```

**Method-1**

**Example:**

```
let arr=[1,2,3,4,"red"]  
arr.forEach((i)=>console.log(i))
```

**Output:**

```
1  
2  
3  
4  
red
```

**Method-2**

```
let arr=[1,2,3,4,"red"]  
arr.forEach(  
    function(i,j){  
        console.log(`The value stored at index ${j} is ${i}`)  
    })
```

**Output:**

```
The value stored at index 0 is 1  
The value stored at index 1 is 2  
The value stored at index 2 is 3  
The value stored at index 3 is 4  
The value stored at index 4 is red
```

**NOTE-**

for..in&for..of are loops in JS  
but forEach is a method

## Comparison of different loops in JavaScript

Feature	For Loop	While Loop	Do-While Loop	For-In	For-Of	forEach
Type	General	General	General	Object keys	Iterable values	Array-specific
Condition Check	Before	Before	After	N/A	N/A	N/A
Usage	Known iterations	Unknown iterations	At least one execution	Objects	Iterables	Arrays
Break Supported	Yes	Yes	Yes	Yes	Yes	No
Return Values	No	No	No	Keys	Values	No
Ideal For	Numbers	Numbers	Numbers	Objects	Arrays, Strings	Arrays

### 1. For Loop

- Purpose:** Iterates a block of code a specific number of times.
- Syntax:**

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

- Example:**

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

- Characteristics:**

Used when number of iterations is known.  
Has explicit control over initialization, condition & iteration.

### 2. While Loop

- Purpose:** Repeats a block of code while a condition is true.
- Syntax:**

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

- Example:**

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

- Characteristics:**

- The condition is checked before the loop body is executed.
- Can result in an infinite loop if the condition never becomes false.

### 3. Do-While Loop

- Purpose:** Executes the loop body at least once and then repeats while the condition is true.
- Syntax:**

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

- Example:**

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

- Characteristics:**

- Ensures the loop body executes at least once, regardless of the condition.
- Condition is checked after executing the loop body.

### 4. For-In Loop

- Purpose:** Iterates over the keys (properties) of an object.
- Syntax:**

```
for (let key in object) {
    // Code to execute }
```

- Example:**

```
const obj = { a: 1, b: 2, c: 3 };
for (let key in obj) {
    console.log(key, obj[key]);
}
```

- Characteristics:**

- Best suited for iterating over objects.
- Iterates over enumerable properties (including inherited ones).
- Should not be used for arrays due to potential issues with indices.



## 5. For-Of Loop

- **Purpose:** Iterates over the values of an iterable (e.g., arrays, strings, maps).

- **Syntax:**

```
for (let value of iterable) {  
    // Code to execute  
}
```

- **Example:**

```
const arr = [1, 2, 3];  
for (let value of arr) {  
    console.log(value);  
}
```

- **Characteristics:**

- Used for iterables like arrays, strings, maps, and sets.
- Does not work with plain objects.

---

## 6. forEach

- **Purpose:** Iterates over elements in an array and applies a callback function to each element.

- **Syntax:** array.forEach(callback);

- **Example:**

```
const arr = [1, 2, 3];  
arr.forEach((value, index) => {  
    console.log(index, value);  
});
```

- **Characteristics:**

- Only works for arrays.
- Does not allow breaking out of the loop.
- Ideal for applying operations to array elements.



# EXERCISE

## Problem-1

- **Program to check whether num is zero,positive or negative**

```
//DYNAMIC INPUT IN node js  
//Install the module prompt-sync  
// npm i prompt-sync  
const prompt = require("prompt-sync")();  
let num = parseInt(prompt("Num 1- "));  
let num=4
```

### //METHOD-1 USING IF-ELSE

```
if(num>0)  
    console.log("positive")  
else if(num<0)  
    console.log("negative")  
else  
    console.log("zero")
```

**Output:** positive

## Problem-2

- **Program to find the minimum of three numbers using dynamic input**

```
const prompt = require("prompt-sync")();  
let a = parseInt(prompt("Num 1- "));  
let b = parseInt(prompt("Num 2- "));  
let c = parseInt(prompt("Num 3- "));  
if (a < b && a < c)  
    console.log(` ${a} is minimum`)  
else if(b < c)  
    console.log(` ${b} is minimum`)  
else  
    console.log(` ${c} is minimum`)
```

### //METHOD-2 USING TERNARY OPERATOR

```
console.log(  
    num>0 ? "positive"  
    : num<0 ? "negative"  
    : "zero"  
)
```

**Output:** positive

### Output:

Num 1- 6  
Num 2- 8  
Num 3- 4  
4 is minimum

## Problem-3

- **Program to take number as input & check whether it's +ve,-ve or zero**

```
const prompt = require("prompt-sync")();  
let a = parseInt(prompt("Enter the number: "));  
if (a == 0)  
    console.log("Zero")  
else if (a > 0)  
    console.log("+ve")  
else  
    console.log("-ve")
```

### Output:

Enter the number: 6  
+ve  
Enter the number: -3  
-ve  
4 is minimum  
Enter the number: 0  
Zero

## Problem-4

- **Program to take 3 numbers as input & print them in descending order**

```
const prompt = require("prompt-sync")();  
let a = parseInt(prompt("Num 1- "));  
let b = parseInt(prompt("Num 2- "));  
let c = parseInt(prompt("Num 3- "));
```

### Output:

Num 1- 6  
Num 2- 3  
Num 3- 9  
9 6 3



```

if (a>b && a>c){
    if (b>c)
        console.log(a,b,c)
    else
        console.log(a,c,b)
}
else if(b>c){
    if (a>c)
        console.log(b,a,c)
    else
        console.log(b,c,a)
}
else{
    if (a>b)
        console.log(c,a,b)
    else
        console.log(c,b,a)
}

```

### Problem-6

- Program to print the sum of factors of a given number

```

tetnum=10 // factor= 1,2,5,10
let sum=0
for(let i=1;i<=num;i++){
    if(num%i==0)
        sum+=i
}
console.log("Sum of factors is-",sum)

```

### Problem-7

- Program to check whether given number is prime or not

#### Method-1

```

let ct=0
for(let i=1;i<=num;i++){
    if(num%i==0)
        ct = ct+1
}
if(ct==2)
    console.log(` ${num} is prime`)
else
    console.log(` ${num} is not prime`)

```

#### By using factorial:

```

const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
let f=1;
for(let i=1;i<=num;i++)
    f=f*i
console.log(`The factorial of ${num} is ${f}`)

```

### Problem-5

- Program to print the factors of a given number

```

const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
//using for loop
for(let i=1;i<=num;i++)
{
    if(num%i==0)
        console.log(i)
}
//using while loop
let i=1
while(i<=num){
    if(num%i==0)
        console.log(i)
    i++
}

```

#### Output:

Sum of factors is- 18

#### Method-2

```

let flag=true
if(num<=1)
    console.log(` ${num} is not prime`)
else{
    for(let i=2;i<num;i++){
        if(num%i==0){
            flag=false
            break
        }
    }
    if(flag)
        console.log(` ${num} is prime`)
    else
        console.log(` ${num} is not prime`)
}

```



### Problem-8

- Find the Gcd (grated common factor) of two numbers

```
const prompt = require("prompt-sync")();
let num1=parseInt(prompt("Enter number1- "))
let num2=parseInt(prompt("Enter number2- "))
while(num2!=0){
    [num1,num2]=[num2,num1%num2]
}
console.log("gcd=",num1)
```

#### Output:

Enter number1- 12  
Enter number2- 22  
gcd=2

### Problem-9

- Write a program to print the Multiplication table

```
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
for(let i=1;i<=10;i++)
    console.log(` ${num} x ${i} = ${num*i}`)
```

#### Output:

Enter a number- 9  
9 x 1 = 9  
9 x 2 = 18  
9 x 3 = 27  
9 x 4 = 36  
9 x 5 = 45  
9 x 6 = 54  
9 x 7 = 63  
9 x 8 = 72  
9 x 9 = 81  
9 x 10 = 90

### Problem-10

- Find the Fibonacci series-- print the terms till n

```
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
let f=-1
let s=1
let c
for(let i=1;i<=num;i++)
{
    c=f+s
    console.log(c)
    f=s
    s=c
}
```

#### Output:

Enter a number- 6  
0  
1  
1  
2  
3  
5

- for nth term in fibonacci series:

```
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter a number- "))
let f=-1
let s=1
for(let i=1;i<=num;i++)
{
    c=f+s
    f=s
    s=c
}
console.log(c)
```

#### Output:

Enter a number- 7  
8

### Problem-11

- Write Program to print the count of each element from given array

```
let arr=[1,2,3,4,3,2,1,2,5,6,4,6,7,8,1]
let d={}
for(let i of arr){
    if(d[i]==undefined)
        d[i]=1
    else
        d[i]+=1
}
console.log(d)
```

#### Output:

Count= { '1': 3, '2': 3, '3': 2, '4': 2, '5': 1, '6': 2, '7': 1, '8': 1 }

### Problem-12

- **Program to print the count of given element from given array**

```
const prompt = require("prompt-sync")();
let arr=[1,2,3,4,3,2,1,2,5,6,4,6,7,8,1]
let ele=parseInt(prompt("Enter the element to be searched- "))
let ct=0
for(let i of arr){
    if(i==ele)
    ct+=1
}
console.log(`The count of ${ele} in given array is ${ct}`)
```

**Output:**

Enter the element to be searched- 6  
The count of 6 in given array is 2

### Problem-13: reverse the elements in a list

```
let arr=[1,2,3,4]
arr.reverse()
console.log(typeof arr, arr)
let ch=arr.toString()
console.log(typeof ch, ch, ch.length)
```

**Output:**  
object [ 4, 3, 2, 1 ]  
string 4,3,2,1 7

### Problem-13

- **reverse a string as per given output**

i/p- "hello to all"

o/p- "all to hello"

```
let s="hello to all"
let r=s.split(" ")
r.reverse()
console.log(r.join(" "))
```

### Problem-15

- **Program to reverse a string as per given output**

i/p- "the sky is blue"

o/p- "the yks is eulb"

```
let s="the sky is blue"
let r=s.split(" ")
for(let i=0;i<r.length;i++){
    if(i%2!=0){

        r[i]=r[i].split("")
        r[i].reverse()
        r[i]=r[i].join("")
    }
}
console.log(r.join(" "))
```

### Problem-14

- **Program to reverse a string as per given output**

i/p- "hello to all"

o/p- "ollehotolleh"

```
let s="hello to all"
let r=s.split(" ")
let n=""
for(let i of r){
    i=i.split("")
    i.reverse()
    n=n+i.join("")+" "
}
console.log(n)
```

# FUNCTIONS

- Function is a block of reusable code that performs a specific task or set of tasks.
- Functions are one of the fundamental building blocks of JavaScript and are used for organizing and structuring your code, making it more modular and easier to maintain.
- A JavaScript function is executed when "something" invokes it (calls it).

## ❖ Advantage of JavaScript function:

- There are mainly two advantages of JavaScript functions.
  1. **Code reusability:** We can call a function several times so it saves coding.
  2. **Less coding:** We don't need to write many lines of code each time to perform a common task.

## Syntax:

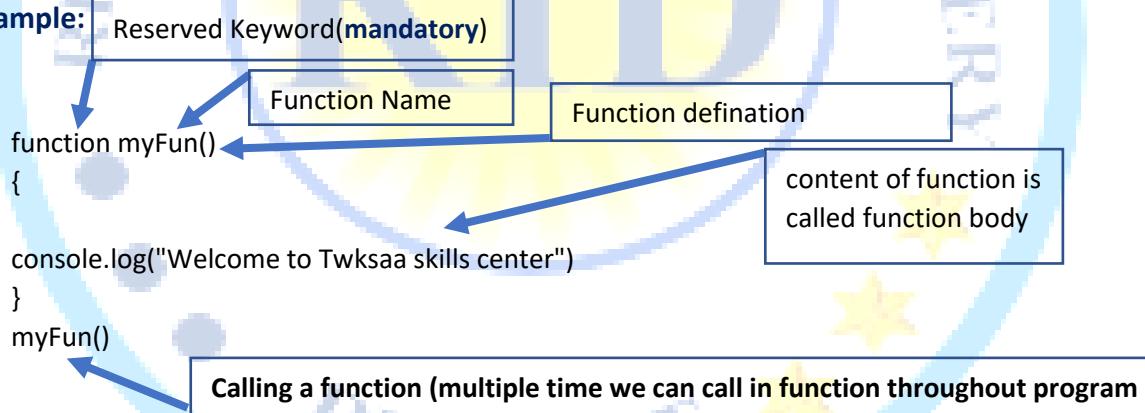
### ➤ **Function Declaration:**

```
function functionName(parameters) {
    // Function body
    // Code to be executed when the function is called
}
```

- **functionName:** This is the name of the function.
- **parameters:** These are the input values (arguments) that the function can accept. You can have zero or more parameters, separated by commas.
- **Function body:** This is where you define code that gets executed when the function is called.

## ❖ User defined Function:

### ❖ Example:



**Output:** Welcome to Twksaa skills center

### ❖ Use of return in function-It is used to store the value returned in function

### ❖ Example-1:

```
function add(a,b){
    let c=a+b ;
    return c ;
}
let sum= add(10,20) ;
console.log("sum of two number=",sum) ;
```

The diagram shows the execution flow of the `add` function:

- a and b are parameters:** `a` and `b` are passed as arguments to the function.
- C return value to add function:** The variable `c` holds the return value of the function, which is the sum of `a` and `b`.
- 10 and 20 are arguments:** The values `10` and `20` are passed as arguments to the `add` function.

**Output:**

sum of two number=30

**Example-2:**

```
function add(a,b)
{
    return a+b
    console.log ("Addition Done")
}
console.log ("sum= ", add (10,20))
let s1=add (40,50)
let s2=add (50, 50)
console.log ("value of s1= ", s1, "value of s2= ", s2)
```

After function return not any statement of code will be execute

Here Multiple time we are calling same add() function with different arguments

**Output:**

sum= 30  
value of s1= 90 value of s2= 100

**Example-3:**

```
function greetStudent(name) {
    console.log(name + " ! Welcome to T3 Skills Center");
}
// Calling the function with different student names
greetStudent("Pooja");
greetStudent("Aarti");
greetStudent("Anjali");
greetStudent("Sohan");
greetStudent("Mohan");
```

**Output:**

Pooja ! Welcome to T3 Skills Center  
Aarti ! Welcome to T3 Skills Center  
Anjali ! Welcome to T3 Skills Center  
Sohan ! Welcome to T3 Skills Center  
Mohan ! Welcome to T3 Skills Center

**Question:**

1. create a function to check the given is prime or not
2. create a function to find the sum natural number
3. create function to check the given number +ve -ve or zero
4. create a function to check given character is uppercase lowercase digit or special symbols
5. create a function to find all vowels in parent in your name or any string
6. Write a program to check if given number is Armstrong number or not

# Function Expression

## (return)

- A JavaScript function can also be defined using an expression.
- A function expression can be stored in a variable with help of return.
- return statement is used to exit a function and return a value to the function caller. When a return statement is executed, the function stops its execution immediately, and the specified value is returned. If no value is specified, undefined is returned by default.

### Syntax:

```
Const functionName = function(parameters) {  
    // Function body  
    // Code to be executed when the function is called  
}
```

- **functionName:** This is an optional name for function. You can omit it if you want an anonymous function.
- **parameters:** These are the input values (arguments) that the function can accept.
- **Function body:** This is where you define the code that gets executed when the function is called.

### Example-1:

```
const add = function(a, b) {  
    return a + b;  
}  
r1=add(15,20)  
console.log(r1)  
r2=add(25,50)  
console.log(r2)  
r3=add(50,53)  
console.log(r3)
```

**Output:**359.

```
75  
103
```

### ❖ Syntax: **return [expression];**

- **expression:** This is optional and represents the value to be returned by the function. It can be any valid JavaScript expression, including variables, constants, objects, arrays, function calls, and more. If omitted, the function returns undefined.

### ❖ Key Points:

- **Exiting a Function:** When a return statement is encountered, the function execution stops immediately, and control is returned to the calling code.
- **Returning Values:** The return statement can return any type of value, including primitives (number, string, boolean, etc.), objects, arrays, and functions.
- **Omitting return:** If a function does not have a return statement, it returns undefined by default.

### Example 1: Returning a Simple Value

```
function add(a, b) {  
    return a + b;  
}  
let result = add(5, 3);  
console.log(result); Output:8
```

### Example-2:

```
const x = function (a, b)  
{  
    return a * b  
}  
let z = x(3, 6)  
console.log(z)
```

### Output:

**Note:** above function is actually an anonymous function (a function without a name). Functions stored in variables do not need function names. They are always invoked (called) using the variable name.



**Note:** function add takes two arguments a and b, and returns their sum using the return statement. The result is then stored in the variable result and printed to the console

**Example 2:** Returning Early

```
function isEven(number) {  
    if (number % 2 === 0) {  
        return true; // Exit the function early if the number is even  
    }  
    return false; // Return false if the number is not even  
}  
  
console.log(isEven(4)); Output:true  
console.log(isEven(7)); Output:false
```

**Example 3:** Returning an Object

```
function createUser(name, age) {  
    return {  
        name: name,  
        age: age  
    };  
}  
let user = createUser("Sangam", 18);  
console.log(user); Output:{ name: 'Sangam', age: 15 }
```

**Example 4:** Implicit Return in Arrow Functions

```
const multiply = (a, b) => a * b;  
console.log(multiply(4, 5)); // Output:20
```

**Note:** arrow function multiply implicitly returns result of a \* b without using the return keyword.

**Example 5:** Returning Undefined

```
function logMessage(message) {  
    console.log(message);  
    return;  
}  
let result = logMessage("Hello, World!");  
console.log(result); Output:undefined
```

**Example 6:** Returning Functions

- Functions can return other functions, allowing for the creation of higher-order functions.

```
function createMultiplier(factor) {  
    return function(number) {  
        return number * factor;  
    };  
}  
const double = createMultiplier(2);  
console.log(double(5));Output:10
```

**Example 7:** Returning Arrays

```
function getArray() {  
    return [1, 2, 3, 4, 5];  
}  
let array = getArray();  
console.log(array); // Output:[1, 2, 3, 4, 5]
```

# Function () Constructor

- It is a built-in function that can be used to create new function objects dynamically.
- It allows to create functions at runtime by providing a list of arguments and a function body as strings.

## Syntax:

**new Function(arg1, arg2, ..., functionBody)**

- **arg1, arg2, ...argN (optional):** Names of the function's arguments as strings.
- **functionBody:** A string representing the JavaScript code to be executed within the function.
- **new keyword** is used to create an instance of an object that has a constructor function.

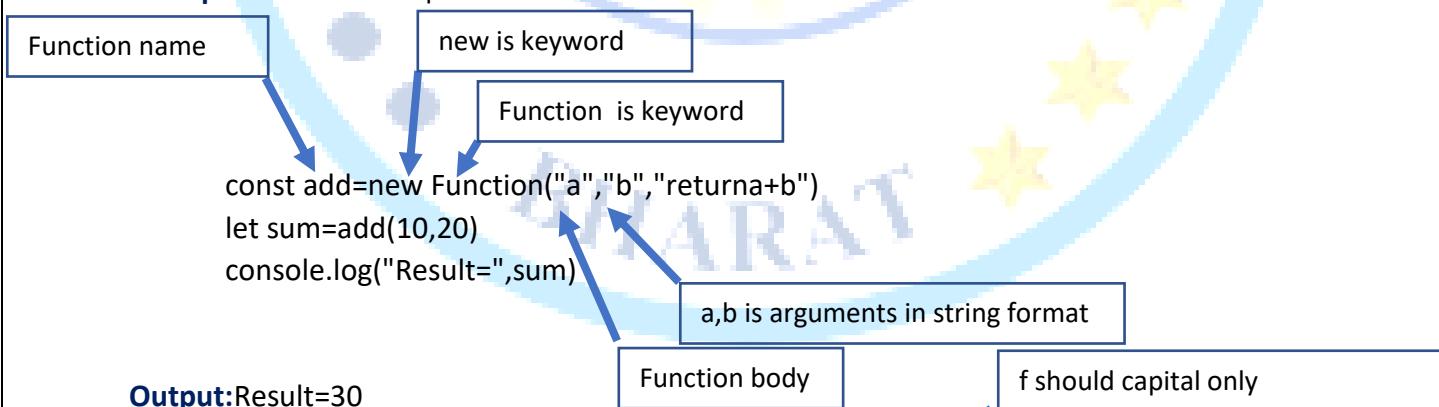
## ❖ Key Points:

- Functions created with Function constructor do not inherit any scope other than global scope.
- These functions are parsed every time they are created, which can impact performance.
- The Function constructor can execute arbitrary code, posing potential security risks, especially with user-generated content.
- When we use new keyword with the Function constructor, it creates a new function object.

## ❖ new Keyword in JavaScript:

- **The new keyword performs the following steps:**
- **Creates a New Object:** A new empty object is created.
- **Sets the Prototype:** The prototype of the new object is set to the prototype property of the constructor function.
- **Binds this:** The constructor function is called with this bound to the new object, allowing properties and methods to be added to it.
- **Returns the Object:** The new object is returned, unless the constructor function explicitly returns a different object. When we use the new keyword with the Function constructor, it creates a new function object.

**Example-1:** Create a simple function that adds two numbers.



## Example-2:

```
// without new keyword
const add= Function("a","b","returna+b") // const add= function("a","b","returna+b")
let sum=add(10,20)
console.log("Result=",sum)
```

**Output:** Result=30



## **Function Parameters and Arguments**

- Function parameters are the names listed in the function definition.
- Function arguments are the real values passed to (and received by) the function.
- Parameters are the placeholders in the function definition, while arguments are the actual values passed when calling the function.

### **Syntax:**

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

### **❖ Parameter Rules:**

- JavaScript function definitions do not specify data types for parameters.
- JavaScript functions do not perform type checking on the passed arguments.
- JavaScript functions do not check the number of arguments received.

### **Example-1:**

```
function greet(name)  
{  
    console.log(`Hello, ${name}!`);  
}  
greet("Sangam Kumar");  
greet();
```

Here name is parameter

Here Sangam Kumar is argument

Calling function without any argument

### **Output:**

```
Hello, Sangam Kumar!  
Hello, undefined!
```

**Note:** If a function is called with missing arguments (less than declared), the missing values are set to undefined.

### **Example-2:**

```
function print(Name,Roll_no,Branch){  
    console.log(`Your Name is ${Name}, Roll_No is: ${Roll_no} and Branch is ${Branch}!`)  
}  
print("Sangam Kumar", 100) // We declare three parameters passing two arguments so that  
// Branch will undefined  
print("Sangam Kumar", 100,"CSE")  
print()
```

### **Output:**

```
Your Name is Sangam Kumar, Roll_No is: 100 and Branch is undefined!  
Your Name is Sangam Kumar, Roll_No is: 100 and Branch is CSE!  
Your Name is undefined, Roll_No is: undefined and Branch is undefined!
```

### **❖ Default Parameters:**

- ES6 allows function parameters to have default values.
- You can specify default values for function parameters, which are used if argument is not provided

### **Example-1:**

```
function greet(name = "Skills") //Here name="skills" is a Default Parameters:
```

```
{
  console.log(`TWKSAA, ${name}`);
}
greet();
greet("Center");
```

**Output:** TWKSAA, Skills!

TWKSAA, Center!

### Example-2:

- If y is not passed or undefined, then y = 15.

```
function myFunction(x, y = 15) {
  return x + y
}
let r= myFunction(20)
console.log(r)
```

**Output:** 35

### ❖ Rest Parameters:

- The rest parameter allows a function to accept an arbitrary number of arguments as an array.
- It is denoted by three dots (...) followed by a parameter name.
- The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

### Example-1:

```
function sum(...numbers) {
  console.log("value=", numbers)
  let s=0
  for (let i in numbers){
    let num=numbers[i]
    console.log(numbers[i])
    s=s+num
  }
  return s// return numbers.reduce((total, num) => total + num, 0)
}
const result = sum(1, 2, 3, 4, 5, 6)
console.log("sum=",result)
```

**Output:** Value= [ 1, 2, 3, 4, 5, 6 ]

```
1
2
3
4
5
6sum= 21
```

### ❖ Arguments are Passed by Value:

- The parameters, in a function call, are the function's arguments.
- JavaScript arguments are passed by value: The function only gets to know the values, not the argument's locations.
- If a function changes an argument's value, it does not change parameter's original value.
- Changes to arguments are not visible (reflected) outside the function.

### Example-3

```
function print(Name,Roll_no,Branch="CSE"){
  console.log(`Your Name is ${Name}, Roll_No is: ${Roll_no} and Branch is ${Branch}!`)
}
print("Sangam Kumar", 100)
print("Sushil Kumar", 100,"CSE")
print("Satyam Kumar", 100,"ME")
print()
```

### Output:

Your Name is Sangam Kumar, Roll\_No is: 100 and Branch is CSE!  
 Your Name is Sushil Kumar, Roll\_No is: 100 and Branch is CSE!  
 Your Name is Satyam Kumar, Roll\_No is: 100 and Branch is ME!  
 Your Name is undefined, Roll\_No is: undefined and Branch is CSE!

### Example-2:

```
function sum(...args) {
  let sum = 0
  for (let n of args){
    sum += n
  }
  return sum
}
let A = sum(3,6, 9, 16, 25, 29, 100, 66, 77,8)
console.log("Sum=",A)
```

**Output:** Sum= 339



**Q1.create a function to check given number is odd or even**

```
function isEven(Number){
    if(Number%2==0){
        // console.log("Enven Number !")
        return "Even Number"
    }
    else{
        // console.log("Odd Number !")
        return "Odd Number"
    }
}
isEven(11)
console.log(isEven(10))
```

**Output:** Even Number !

**Q3.Create a function to check a person is eligible for vote or not**

```
let age=Number(prompt("Enter your Age: "))
function vote(n){
    if(n>=18){
        console.log("your are eligible for vote")
        // return "your are eligible for vote"
    }
    else{
        console.log("not")
        // return "not"
    }
}
isEven(age)
console.log(isEven(age))
```

**Q2.create a function find the sum of all even number**

```
let num=Number(prompt("Enter the number : "))
function natural_nu_sum(n){
    let sum=0
    for (let i=1; i<=n; i++){
        console.log(i)
        sum=sum+i
    }
    console.log("sum of natural number=",sum)
    // return "sum of natural number=" +sum
}
natural_nu_sum(num)
// console.log(natural_nu_sum(num))
```

**Q4. Create a function to find sum of natural number**

```
const prompt=require('prompt-sync')()
let num=Number(prompt("Enter the number : "))
function natural_nu_sum(n){
    let sum=0
    for (let i=1; i<=n; i++){
        console.log(i)
        sum=sum+i
    }
    console.log("sum of natural number=",sum)
    // return "sum of natural number=" +sum
}
natural_nu_sum(num)
// console.log(natural_nu_sum(num))
```

**Q5.Create a function to display the district according to the state name**

```
const prompt = require('prompt-sync')();
// Define district functions
function bihar_dist() {
    console.log("Bihar Districts:");
    console.log("1. Rohtas");
    console.log("2. Patna");
    console.log("3. Betiya");
    console.log("4. ARA");
}

function mp_dist() {
    console.log("Madhya Pradesh Districts:");
    console.log("1. Bhopal");
    console.log("2. Indore");
    console.log("3. Gwalior");
    console.log("4. Jabalpur");
}
```

```
// Main state function
function state() {
    const states = {
        "bihar": bihar_dist,
        "madhya pradesh": mp_dist
    };
    let st = prompt("Enter your state name: ").toLowerCase();
    let found = false;
    for (let key in states) {
        if (key === st) {
            states[key](); // Call the respective district function
            found = true;
            break;
        }
    }
    if (!found) {
        console.log("Invalid state. Please try again.");
    }
}
state(); // Call the state function
```



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"><title>Document</title>
</head>
<body>
    <label>Number</label>
    <input type="text" id="d1" autocomplete="off">
    <button type="button" onclick="isEven()">Even</button>
    <button type="button" onclick="isPositive()">Positive</button>
    <button type="button" onclick="isFactor()">Factor</button>
    <button type="button" onclick="isPrime()">Prime</button>
    <button type="button" onclick="isNutralNumSum()">Natural
    Sum</button>
    <button type="button" onclick="isSquare()">Square</button>
    <button type="button" onclick="isPalindrome()">Palindrome</button>
    <button type="button" onclick="isArmstrong()">Armstrong</button>
    <button type="button" onclick="isClear()">Clear</button>
    <button type="button" onclick="isCopy()">Copy</button>
    <button type="button" onclick="isCut()">Cut</button> <br>
    <label>Result</label>
    <input type="text" id="res1"> <input type="text" id="res2">
    <script>
        function isEven() {
            let num = Number(document.getElementById("d1").value);
            if (num % 2 === 0) {
                document.getElementById("res1").value = "Even Number";
                document.getElementById("res2").value = num;
            } else {
                document.getElementById("res1").value = "Odd Number";
                document.getElementById("res2").value = num;
            }
        }

        function isPositive() {
            let num = Number(document.getElementById("d1").value);
            if (num > 0) {
                document.getElementById("res1").value = "Positive Number";
                document.getElementById("res2").value = num;
            } else if (num < 0) {
                document.getElementById("res1").value = "Negative Number";
                document.getElementById("res2").value = num;
            } else {
                document.getElementById("res1").value = "Zero";
                document.getElementById("res2").value = num;
            }
        }

        document.getElementById("res1").value = "Negative num";
        else {
            let Result = "";
            let c = 0;
            for (let i = 1; i <= num; i++) {
                if (num % i === 0) {
                    Result = Result + i + ",";
                    c = c + 1;
                }
            }
            document.getElementById("res1").value = Result;
            document.getElementById("res2").value = "Count=" + c;
        }

        function isPrime() {
            let num = Number(document.getElementById("d1").value);
        }
    </script>

```

```

        let c = 0;
        if (num <= 0) {
            document.getElementById("res1").value = "Not Prime";
        } else {
            for (let i = 1; i <= num; i++) {
                if (num % i === 0) {
                    c = c + 1;
                }
            }
            if (c == 2) {
                document.getElementById("res1").value = "Prime Number";
                document.getElementById("res2").value = num;
            } else {
                document.getElementById("res1").value = "Not Prime Number";
            }
        }
        document.getElementById("res2").value = num;
    }

    function isNutralNumSum() {
        let num =
            Number(document.getElementById("d1").value);
        let sum = 0;
        for (let number = 1; number <= num; number++) {
            sum = sum + number;
        }
        document.getElementById("res1").value = "sum of natural
        numbar";
        document.getElementById("res2").value = sum;
    }

    function isSquare() {
        let num =
            Number(document.getElementById("d1").value);
        let s = Math.pow(num, 2);
        console.log(s);
        document.getElementById("res1").value = s;
        document.getElementById("res2").value = num;
    }

    function isPalindrome() {
        let num =
            Number(document.getElementById("d1").value);
        if (isNaN(num)) {
            alert("Please enter a valid number");
            return;
        }
        let originalNum = num;
        let reversedNum = 0;
        while (num > 0) {
            let lastDigit = num % 10;
            reversedNum = reversedNum * 10 + lastDigit;
            num = Math.floor(num / 10);
        }
        if (reversedNum === originalNum) {
            document.getElementById("res1").value = "Palindrome
            Number";
            document.getElementById("res2").value = reversedNum;
        } else {
            document.getElementById("res1").value = "Not a
            Palindrome Number";
            document.getElementById("res2").value = reversedNum;
        }
    }

```



```

function isArms() {
let num = Number(document.getElementById("d1").value);
if (isNaN(num) || num <= 0) {
    alert("Please enter a valid positive number");
    return;
}
let originalNum = num;
let len = String(num).length;
let updateNum = 0;
while (num > 0) {
    let lastDigit = num % 10;
    updateNum = updateNum + Math.pow(lastDigit, len);
    num = Math.floor(num / 10);
}
if (originalNum === updateNum) {
    document.getElementById("res1").value = "Armstrong Number";
    document.getElementById("res2").value = updateNum;
} else {
    document.getElementById("res1").value = "Not an Armstrong Number";
    document.getElementById("res2").value = updateNum;
}
function iscopy() {
    let sonu = document.getElementById("d1")
    sonu.select()
    document.execCommand("copy")
}
function isCut() {
    let t = document.getElementById("res1")
    t.select()
    document.execCommand("cut")
    document.getElementById("res1") = " "
}
function isClear() {
    document.getElementById("d1").value = " "
    document.getElementById("res1").value = " "
    document.getElementById("res2").value = " "
}
</script>
</body>
</html>

```

Number  Even Positive Factor Prime Natural Sum Square Palindrome Armstrong Clear Copy Cut

Result Even Number

# ARROW FUNCTION

- Arrow functions provide a concise way to write functions, especially for simple one-liners. They automatically return the expression result. Arrow functions, introduced in ES6 (ECMAScript 2015)

## ❖ Key Features:

- Short Syntax:** Arrow functions are more concise compared to traditional function expressions.
- Lexical this:** They do not have their own this; instead, they inherit it from the surrounding function or context.
- No arguments object:** Arrow functions do not have their own arguments object.
- Cannot be used as constructors:** Arrow functions cannot be used with the new keyword.

## Syntax:

arrow function is defined using the => (fat arrow)

```
const functionName = (parameters) => {  
    // Function body  
    // Code to be executed when the function is called  
}  
functionName() // function call
```

- FunctionName:** This is an optional name for the function, just like in function expressions.
  - parameters:** These are the input values (arguments) that the function can accept.
  - Function body:** This is where you define the code that gets executed when the function is called.
- **Arrow function {}=>** is a concise way of writing Javascript functions in a shorter way. Arrow functions were introduced in the ES6 version.

## Method-1

```
const add3=(a,b)=>console.log(a+b)  
add3(10,20)
```

**Output:** 30

## Method-2

```
let add3=(x,y)=>x+y // Here add3 is function name &x,y are parameters  
console.log(add3(10,20))  
Recursion-When a function calls itself, it is recursion
```

## Example-1:

### • Arrow function without parameters

```
const greet = () => {  
    console.log("Hello!");  
}
```

### • Arrow function with a single parameter

```
const square = (x) => {  
    return x * x;  
}
```

### • Arrow function with multiple parameters

```
const add = (a, b) => {  
    return a + b;  
}
```

### • Arrow function with a concise body (implicit return)

```
const double = (x) => x * 2;
```

### • How to pass multiple arguments in single parameters

#### Example:

```
const add= (...n)=>{  
    for(let i of n){  
        console.log(i)  
    }  
}  
add(10,20,30,40,50)
```

## Example-2:

```
function Person(name) {  
    this.name = name;
```



#### Regular function with a different "this" context

```
this.sayHello = function () {  
    console.log(`Hello, ${this.name}!`);  
};
```

#### Arrow function with the same "this" context

```
this.sayHi = () => {  
    console.log(`Hi, ${this.name}!`);  
}  
  
const person = new Person("Sangam Kumar");  
person.sayHello(); // Outputs "Hello, Sangam Kumar!"  
person.sayHi(); // Outputs "Hi, Sangam Kumar!"
```

**Output:** Hello, Sangam Kumar!

Hi, Sangam Kumar!

#### ❖ Implicit Return:

- Arrow functions allow for implicit return when the function body consists of a single expression. In such cases, you can omit the curly braces {} and the return keyword. The result of the expression is automatically returned.

**Example:** const square = (x) => x \* x; // Implicit return

```
const add = (a, b) => {  
    return a + b; // Explicit return};
```

#### ❖ When to Use Arrow Functions:

- Arrow functions are most suitable for short, concise functions, especially those used as callbacks or for simple calculations.
- They are not a replacement for regular functions in all scenarios, as regular functions still have their place when need flexibility of this keyword, function hoisting, or named functions.

#### Example-1: Program to find the all even number from a given list by using arrow function

```
const list1=[10,20,30,4,75,4,7,55,4,7,5,204,4,5,21]  
let even_no=""  
const even=(num)=>{  
    for(let n in num){  
        // console.log(n)  
        if (num[n]%2==0){  
            even_no+=num[n]+", "  
        }  
    }  
    even(list1)  
    console.log("Even Number=",even_no)
```

**Output:** Even Number= 10, 20, 30, 4, 4, 4, 204, 4

**// Create a arrow function to count the all vowels are parent in given string**

```
const prompt=require('prompt-sync')()  
let s=prompt("Enter the string: ")  
const vowels=(n)=>{  
    let c=0, let s=""  
    for (let char of n){  
        if(char=="a" || char=="e" || char=="i" ||  
        char=="o" || char=="u"){  
            // console.log(char)  
            c=c+1  
            s=s+" "+char  
        }  
    }  
    console.log("count=",c)  
    console.log("vowles=",s)  
} vowels(s)  
PS C:\Users\hp\Desktop\RID ONLINE\ > node funct4.js  
Enter the string: RID Bharat Bhopal  
count= 4  
vowles= a a o a
```



# **FUNCTION INVOCATION(CALL)**

- function invocation is the process of calling a function to execute its code.
- To execute a function, you need to invoke (call) it by using its name followed by parentheses.
- If the function has parameters, you pass arguments inside the parentheses.

## **❖ Why function invocation needs?**

- The code inside a function is not executed when the function is defined.
- The code inside a function is executed when the function is invoked.
- It is common to use the term "call a function" instead of "invoke a function".

## **❖ How many ways you can call the function or function is invoked?**

- The following ways you can call a function.
  - 1) Function Declaration
  - 2) Function Expression
  - 3) Arrow Function
  - 4) Method Invocation
  - 5) Constructor Invocation
  - 6) Function.call()
  - 7) Function.apply()
  - 8) Function Invocation without Arguments
  - 9) Immediately Invoked Function Expression (IIFE)
  - 10) Callback Functions

## **1. Function Declaration:**

- You can define a function using the 'function' keyword and then call it by its name.

### **Example:**

```
function greet(name) {  
    console.log(`Hello, ${name}!`);  
}  
greet("Sangam Kumar"); // Calling a function declared using function declaration
```

**Output:** Hello, Sangam Kumar!

## **2. Function Expression:**

- You can define a function as an expression and then call it. This is often used for anonymous functions or functions assigned to variables.

### **Example:**

```
const sayHello = function (name) {  
    console.log(`Hello, ${name}!`);  
};  
sayHello("Satyam");  
// Calling a function defined as an expression
```

**Output:** Hello, Satyam!

```
functionprint(a,b){  
returna+b  
}  
lets=print(10,20)  
console.log(s)
```

## **3. Arrow Function:**

- Arrow functions provide a concise way to define and call functions.

### **Example:**

```
const multiply = (a, b) =>  
{  
    let m= a * b;
```



```
        return m
    }
const result = multiply(4, 5); // Calling an arrow function
console.log("Multiplication=",result)
```

**Output:**Multiplication= 20

#### **4. Method Invocation:**

- When a function is a property of an object, you can invoke it using dot notation.

**Example:**

```
const person = {
    name: "Sangam",
    greet: function () {
        console.log(`Hello, ${this.name}!`);
    },
};
person.greet(); // Calling a method of an object
```

**Output:**Hello, Sangam!

#### **5. Constructor Invocation:**

- You can create instances of objects using constructor functions and the `new` keyword.

**Example:**

```
const greet = new Function("name", "title", "return name + ' ' + title;");
const result = greet("Sangam", "Kumar");
console.log(result);
```

**Output:**Sangam Kumar

#### **6. Function.call() and Function.apply():**

- These methods allow you to explicitly invoke a function and specify the value of `this` and pass arguments.

**Example:**

```
function introduce() {
    console.log(`Hello, I am ${this.name}.`);
}
const person = { name: "Sangam" };
introduce.call(person); // Using call
introduce.apply(person); // Using apply
```

**Output:**

```
Hello, I am Sangam.  
Hello, I am Sangam.
```

#### **7. Function Invocation without Arguments:**

- Functions can also be invoked without passing any arguments by using empty parentheses `()`.

**Example:**

```
function color(){
    console.log("Blue is best color")
}
color()
```

**Output:**Blue is best color

## **8. Immediately Invoked Function Expression (IIFE):**

- IIFE is a function that is defined and invoked immediately. It's often used to create a private scope for variables.

```
(function () {  
    // Code here  
})();
```

### **Example:**

```
(function () {  
    // Private variables and functions  
    let name = "Sangam Kumar";  
    let age = 12;  
    function greet() {  
        console.log(`Hello, my name is ${name} and I am ${age} years old.`);  
    }  
    // Invoking the greet function within the IIFE  
    greet();  
})();  
// Trying to access name and greet outside the IIFE will result in an error  
// console.log(name); // Uncaught ReferenceError: name is not defined  
// greet(); // Uncaught ReferenceError: greet is not defined
```

**Output:** Hello, my name is Sangam Kumar and I am 12 years old.

## **9. Callback Functions:**

- You can pass functions as arguments to other functions and call them later.

### **Example:**

```
function performOperation(x, y, operation) {  
    return operation(x, y);  
}  
function add(a, b) {  
    return a + b;  
}  
const result1 = performOperation(3, 4, add); // Calling a function as a callback  
const result2 = add(3, 4); // Calling the add function  
console.log("sum=", result1)  
console.log("sum=", result2)
```

**Output:** sum= 7, sum= 7

- Return Statement:
- Functions can return values using the return statement. The returned value can be used wherever the function is called. If a function doesn't specify a return value, it implicitly returns undefined.

### **Example:**

```
function subtract(a, b) {  
    return a - b;  
}  
const difference = subtract(8, 3); // difference will be 5
```

### **❖ Function Scope:**

- Variables declared inside a function are locally scoped, meaning they are only accessible within that function. This concept is known as "function scope."



**Example:**

```
function scopeExample() {  
    const localVar = "I am local";  
    console.log(localVar); // Accessible within the function  
}  
scopeExample();  
console.log(localVar); // Error: localVar is not defined
```

❖ **Function Hoisting:**

- Function declarations are hoisted, meaning they can be called before they are defined in the code. This is not the case with function expressions.

**Example:**

```
hoistedFunction(); // This works  
function hoistedFunction() {  
    console.log("I was hoisted.");  
}  
nonHoistedFunction(); // Error: nonHoistedFunction is not a function  
const nonHoistedFunction = function () {  
    console.log("I was not hoisted.");  
};
```

❖ **Closures:**

- A closure is a function that remembers the variables from the outer (enclosing) function's scope even after the outer function has finished executing. Closures are often used to create private variables and maintain state.

**Example:**

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

❖ **Function Declarations vs. Expressions:**

- Function declarations are hoisted and can be called before they are defined. Function expressions are not hoisted and can only be called after they are defined.

```
// Function Declaration  
hoistedFunction(); // Works  
function hoistedFunction() {  
    console.log("I was hoisted.");  
}  
// Function Expression  
nonHoistedFunction(); // Error: nonHoistedFunction is not a function  
const nonHoistedFunction = function () {  
    console.log("I was not hoisted.");  
};
```



## Higher-Order Functions

- Higher-order function is a function that takes one or more functions as arguments or returns a function as its result. Higher-order functions are a powerful concept in JavaScript, enabling functional programming techniques.

### **Example-1:**

```
const ab = () => console.log("I am ab function data");
const bc = () => console.log("This is bc function data");
function msg(a, b) {
    a();
    b();
    function add() {
        let c=10+20
        console.log("sum of two number=",c)
        console.log("Inside the add function");
    }
    return add;
}
msg(ab, bc);
const raj = msg(ab, bc);
console.log(raj);
raj();
```

#### **Output:**

```
I am ab function data
This is bc function data
I am ab function data
This is bc function data
[Function: add]
Inside the add function
```

### **Example-2:**

```
function greet(fn) { // Higher-order function
    console.log("Hello! I'm about to call your function...");
    fn();
}
function sayHi() {
    console.log("Hi, this is a simple message!");
}
greet(sayHi);
```

### **Example-3:**

```
function multiplyBy(num) {
    return function (x) {
        return x * num;
    };
}
const double = multiplyBy(2); // Returns a function that doubles numbers
const triple = multiplyBy(3); // Returns a function that triples numbers
console.log(double(5)); // 10
console.log(triple(5)); // 15
```



## **Anonymous Functions**

- Functions that don't have a name are called anonymous functions. They are often used as arguments to other functions or as immediately invoked function expressions (IIFE).

**Example:**

```
const greeting = function (name) {  
    console.log(`Hello, ${name}!`);  
};  
greeting("Sangam");  
// IIFE (Immediately Invoked Function Expression)  
(function () {  
    console.log("I am an IIFE.");  
})();
```

❖ **Callbacks and Asynchronous JavaScript:**

- JavaScript commonly uses callbacks to handle asynchronous operations like reading files or making network requests. A callback is a function passed as an argument to another function and is executed later when a specific event occurs.

**Example:**

```
function fetchData(callback) {  
    setTimeout(() => {  
        const data = "Some async data";  
        callback(data);  
    }, 1000);  
}  
function process(data) {  
    console.log(`Processing: ${data}`);  
}  
fetchData(process); // Executes process when data is ready
```

❖ **Function Scopes:**

- JavaScript has two main types of function scope: global scope (accessible from anywhere) and local scope (limited to the function where a variable is declared).

```
// Global scope  
const globalVar = "I am global";  
function scopeExample() {  
    // Local scope  
    const localVar = "I am local";  
    console.log(globalVar); // Accessible in local scope  
}  
scopeExample();  
console.log(globalVar)
```

## Callback Function

- **callback function** is a function that is passed as an argument to another function and is executed later, either after the completion of an asynchronous operation or as part of a specific flow.

### Example 1: Synchronous Callback

- In this example, the callback is executed immediately within the function it is passed to.

```
function greet(name, callback) {  
    console.log(`Hello, ${name}!`);  
    callback();  
}  
  
function sayGoodbye() {  
    console.log("Goodbye!");  
}  
  
greet("Ankit", sayGoodbye);
```

**Output:**

Hello, Ankit!  
Goodbye!

### Example 2: Asynchronous Callback (Using setTimeout)

- Callbacks are often used with asynchronous operations like setTimeout.

```
function fetchData(callback) {  
    console.log("Fetching data...");  
    setTimeout(() => {  
        console.log("Data fetched!");  
        callback();  
    }, 2000);  
}  
  
function processData() {  
    console.log("Processing data...");  
}  
  
fetchData(processData);
```

**Output:**

Fetching data...  
Data fetched!  
Processing data...

### Example 3: Callback with Array Methods

- Array methods like forEach, map, filter, etc., use callbacks to process each element.

```
const numbers = [1, 2, 3, 4, 5];  
numbers.forEach((number) => {  
    console.log(number);  
});
```

**Output:**

1  
2  
3  
4  
5

### Example 4: Callback with Event Listeners

- Callbacks are commonly used in event handling, such as listening to user actions.

```
// HTML: <button id="clickMe">Click Me</button>  
const button = document.getElementById("clickMe");  
button.addEventListener("click", () => {  
    console.log("Button was clicked!");  
});
```

**Output (when the button is clicked):**

Button was clicked!

### Example 5: Custom Higher-Order Function

- You can create your own higher-order function that accepts a callback.

```
function calculate(a, b, operation) {
    console.log("Calculating...");
    return operation(a, b); // Execute the callback function
}

const add = (x, y) => x + y;
const multiply = (x, y) => x * y;
console.log(calculate(3, 4, add)); // Output: 7
console.log(calculate(3, 4, multiply)); // Output: 12
```

#### Key Takeaways:

- Definition:** A callback is a function passed as an argument to another function.
- Use Cases:**
  - Handling asynchronous operations (setTimeout, API calls, etc.).
  - Processing array elements (forEach, map, etc.).
  - Event handling (addEventListener).
- Examples in Real Life:**
  - User clicks a button -> Execute a callback to handle the click.
  - Fetching data from a server -> Execute a callback after receiving the data.

Callbacks allow you to design flexible, reusable, and asynchronous code in JavaScript.

### Problem-1

#### ➤ Program to find nth prime number using function

```
function isPrime(n){
    let c=0
    for(let i=1;i<=n;i++){
        if(n%i==0)
            c=c+1
    }
    if(c==2)
        return true
    else
        return false
}
const prompt = require("prompt-sync")();
let num=parseInt(prompt("Enter number- "))
let c=0
let pn=2
while(c<=num){
    if(isPrime(pn)){
        c=c+1
        if(c==num)
            console.log("nth Prime Number=",pn)
    }
    pn+=1 }
```

#### Output:

```
Enter number- 5
nth Prime Number=11
```

### Problem-2:

```
function isPrime(number) {
    if (number <= 1) {
        return false;
    }
    for (let i = 2; i <= Math.sqrt(number); i++) {
        if (number % i === 0) {
            return false;
        }
    }
    return true;
}
console.log(isPrime(2)); // true
console.log(isPrime(3)); // true
console.log(isPrime(4)); // false
console.log(isPrime(17)); // true
```



## Differences between Normal Functions, Arrow Functions, Higher-Order Functions, Anonymous Functions, and Callback Functions in JavaScript

### 1. Normal Function

- A **normal function** (or traditional function) is defined using the function keyword and can have a name. It supports the use of this in an object context.

#### Syntax:

```
function normalFunction(a, b) {  
    return a + b;  
}  
console.log(normalFunction(2, 3)); // Output: 5
```

#### Characteristics:

- Defined using the function keyword.
- Can use the this keyword, bound to the object calling it.
- Can be **hoisted**, meaning you can call it before defining it in the code.

### 2. Arrow Function

- An **arrow function** is a shorter syntax for writing functions. It does not have its own this or arguments context, making it ideal for callbacks or simpler functions.

#### Syntax:

```
const arrowFunction = (a, b) => a + b;  
console.log(arrowFunction(2, 3)); // Output: 5
```

#### Characteristics:

- Uses the => syntax.
- Does **not** bind this; it inherits this from its surrounding context (lexical scoping).
- Cannot be hoisted.
- Concise syntax; can omit {} for single-expression functions.

### 3. Higher-Order Function

- A **higher-order function** is a function that either:
  - Takes another function as an argument.
  - Returns a function.

#### Syntax:

```
function higherOrderFunction(fn) {  
    console.log("Calling the passed function:");  
    fn(); }
```

#### Characteristics:

- Useful in functional programming.
- Examples: map, filter, and reduce in JavaScript arrays.

#### Example:

```
function sayHello() {  
    console.log("Hello!");  
}  
// Passing `sayHello` to  
// `higherOrderFunction`  
higherOrderFunction(sayHello);
```

### 4. Anonymous Function

- An **anonymous function** is a function that does not have a name. It is typically used where functions are passed as arguments or assigned to variables.

#### Syntax:

```
const anonFunction = function (a, b) {  
    return a + b;  
}; console.log(anonFunction(2, 3)); // Output: 5
```

#### Characteristics:

- No name; often used as a value (e.g., assigned to variables, passed as arguments).
- Commonly used in callbacks or inline functions.



## 5. Callback Function

- callback function is a function passed as an argument to another function and executed later.

### Syntax:

```
function processUserInput(callback) {
    console.log("Processing user input...");
    callback();
}
```

### Characteristics:

- Passed as an argument to another function.
- Often used in asynchronous operations, like setTimeout or API calls.

### Example:

```
function displayMessage() {
    console.log("Hello, User!");
} // Passing `displayMessage` as a
callback
processUserInput(displayMessage);
```

Feature	Normal Function	Arrow Function	Higher-Order Function	Anonymous Function	Callback Function
Definition	Standard <code>function</code> keyword	Uses <code>=&gt;</code> syntax	Takes or returns a function	No name	Passed to another function
this Binding	Dynamic (based on caller)	Lexical (from outer scope)	Depends on implementation	Same as normal or arrow	Same as normal or arrow
Hoisting	Yes	No	N/A	No	N/A
Usage	General-purpose functions	Shorter syntax, callbacks	Functional programming	Inline or short functions	Passed to be executed later
Example	<code>function fn() {}</code>	<code>const fn = () =&gt; {}</code>	<code>fn(() =&gt; {})</code>	<code>() =&gt; {}</code> or <code>function()</code>	<code>fn(callback)</code>



## Synchronous and Asynchronous in javascript

- Synchronous and Asynchronous programming describe how code is executed, especially when handling tasks that may take time, like reading a file, making an API call, or setting a timer.

### ❖ Synchronous Programming

#### 1. Definition:

- In synchronous programming, tasks are executed **one after another** in the order they appear in the code.
- A task must complete before the next one starts, which means **blocking** the execution of subsequent code until the current task is finished.

#### 2. Characteristics:

- Code runs in a single thread.
- Tasks are executed sequentially.
- A time-consuming task (like a loop or large computation) will block further code execution.

#### 3. Example:

```
console.log("Start");
function longTask() {
    for (let i = 0; i < 1e9; i++) {} // Simulating a long task
```

### Output:

```
Start
Long task finished
End
```



```
        console.log("Long task finished");
    }
    longTask(); // Blocks the execution
    console.log("End");
```

- Here, the longTask blocks the code execution, so "End" is only logged after it finishes.

## ❖ Asynchronous Programming

### 1. Definition:

- In asynchronous programming, tasks can be executed **out of order**. The program doesn't wait for a task to finish before moving to the next one.
- Time-consuming tasks (like file reads, API calls, or timers) are handled in the background, and when they complete, a **callback**, **Promise**, or **async/await** mechanism is used to handle the result.

### 2. Characteristics:

- Code runs non-blocking; long tasks don't prevent the execution of subsequent code.
- Often used for I/O operations, such as API calls, file access, and timers.
- JavaScript uses the **Event Loop** to manage asynchronous tasks.

### 3. Example with setTimeout:

```
console.log("Start");
setTimeout(() => {
    console.log("Asynchronous task finished");
}, 2000); // Simulates a 2-second delay
console.log("End");
```

**Output:**  
Start  
End  
Asynchronous task finished

- Here, the setTimeout is asynchronous. It doesn't block the program and allows "End" to log while waiting for 2 seconds.

## Key Differences

Feature	Synchronous	Asynchronous
Execution	Tasks are executed in order.	Tasks can run concurrently.
Blocking	Code execution is blocked.	Code execution is non-blocking.
Use Case	Simple tasks.	Time-consuming tasks like API calls.
Examples	Loops, mathematical operations.	Timers, AJAX, file reading.



# **OBJECT**

- Object is data types that store value in key value pair.
- In JavaScript, almost "everything" is an object.
  - Booleans can be objects (if defined with the new keyword)
  - Numbers can be objects (if defined with the new keyword)
  - Strings can be objects (if defined with the new keyword)
  - Dates are always objects
  - Maths are always objects
  - Regular expressions are always objects
  - Arrays are always objects
  - Functions are always objects
  - Objects are always objects
  - All JavaScript values, except primitives, are objects.
- JavaScript object is an entity having state and behavior (properties and method).

**Example:** car, Mobile, pen etc.

- JavaScript is an object-based language. Everything is an object in JavaScript.
- JavaScript is template based not class based. (Means don't create class to get the object. But, we directly create objects.)

**Note:** Objects written as name value pairs are similar to:

- Associative arrays in PHP
- Dictionaries in Python
- Hash tables in C
- Hash maps in Java
- Hashes in Ruby and Perl

## **❖ Object Methods:**

- Methods are actions that can be performed on objects.
- Object properties can be both primitive values, other objects, and functions.
- An object method is an object property containing a function definition.

## **❖ Creating Objects:**

- There are following ways to create objects.
  1. **Object Literal:** Using {} to define an object with properties and values.
  2. **Constructor Functions:** Creating objects using functions with the new keyword.
  3. **ES6 Class Syntax:** Using the class syntax introduced in ECMAScript 2015 (ES6).
  4. **Object.create():** Creating objects with a specified prototype using Object.create().
  5. **Factory Functions:** Creating objects with regular functions that return objects.
  6. **Singleton Objects:** Creating a single instance of an object throughout the application.
  7. **Using the new Operator with Built-in Constructors:** Creating objects of built-in types like Array, Date, and RegExp using new.

### **Example:**

#### **1.Object Literal:**

```
let person = {  
    firstName: "Sangam",  
    lastName: "Kumar",  
};
```

#### **2.Constructor Functions:**

```
function Person(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName; }  
let person = new Person("Satyam",  
    "Kumar");  
console.log(person)
```

#### **3.ES6 Class Syntax:**

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    } }  
let person = new Person("Sangam", "Kumar");
```



#### 4.Object.create():

```
let personPrototype = {  
    firstName: "",  
    lastName: "", };  
let person = Object.create(personPrototype);  
person.firstName = "Sangam";  
person.lastName = "Kumar";
```

#### 5. Factory Functions:

```
function createPerson(firstName, lastName) {  
    return {  
        firstName: firstName,  
        lastName: lastName,  
    };  
}  
let person = createPerson("Sangam", "Kumar");
```

### 1.Object Literal:

- The simplest and most commonly used way to create an object is by using the object literal syntax. You define an object and its properties within curly braces {}.

#### Example:

```
let person = { // Creating an object using Object Literal  
    firstName: "Sangam",  
    lastName: "Kumar",  
    age: 15, } // Accessing properties of the object  
console.log(person.firstName)  
console.log(person.lastName)  
console.log(person.age)
```

#### Output:

```
Sangam  
Kumar  
15
```

**Note:** in this example, we've created an object named person with three properties: firstName, lastName, and age.

### 2.Constructor Functions:

- You can create objects in JavaScript using constructor functions.
- Constructor functions are regular functions that are intended to be used with the **new keyword** to create instances of objects.

#### Example:

```
function Person(firstName, lastName, age) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
} // Creating instances of the Person object using the constructor  
let person1 = new Person("Sangam", "Kumar", 15);  
let person2 = new Person("Satyam", "Kumar", 20);  
// Accessing properties of the objects  
console.log(person1.firstName);  
console.log(person1.lastName);  
console.log(person1.age);  
console.log(person2.firstName);  
console.log(person2.lastName);  
console.log(person2.age);
```

#### Output:

```
Sangam  
Kumar  
15  
Satyam  
Kumar  
20
```

#### Explanation:

- We define a constructor function Person with parameters firstName, lastName, and age.
- Inside the constructor function, we use the **this keyword to assign values to the object's properties based on the provided arguments.**
- We create two instances of the Person object, person1 and person2, using the **new keyword followed by the constructor function.**

### 3.ES6 Class Syntax:

- You can create objects in JavaScript using ES6 class syntax.
- Classes provide a more structured and familiar way to define object blueprints with constructors and methods

#### Example:

```
class Person {  
    constructor(firstName, lastName, age) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
    sayHello() {  
        console.log(`Hello, my name is ${this.firstName}  
        ${this.lastName}.`);  
    }  
} // Creating instances of the Person class  
let person1 = new Person("Sangam", "Kumar", 15);  
let person2 = new Person("Satyam", "Kumar", 20);  
// Accessing properties and calling a method of the objects  
console.log(person1.firstName);  
console.log(person1.lastName);  
console.log(person1.age);  
console.log(person2.firstName);  
console.log(person2.lastName);  
console.log(person2.age);  
person1.sayHello();  
person2.sayHello();
```

```
class SectionA{  
    constructor(Teche, Girls, Boys, Madam, sir){  
        this.a=Teche  
        this.b=Girls  
        this.c=Boys  
        this.d=Madam  
        this.e= sir  
    } }  
let jsclass=new SectionA("Abhishek",  
"Ritisha", "Kunal", "Nandni", "Gunjan")  
// console.log(jsclass)  
console.log(jsclass.a)  
console.log(jsclass.d)  
console.log(jsclass.e)
```

#### Output:

```
Sangam  
Kumar  
15  
Satyam  
Kumar  
20  
Hello, my name is Sangam Kumar.  
Hello, my name is Satyam Kumar.
```

#### Explanation:

- We define a class Person using the **class** keyword, which includes a constructor method to initialize object properties and a **sayHello** method.
- Inside the constructor method, we use the **this** keyword to assign values to the object's properties based on the provided arguments.
- We create two instances of the Person class, **person1** and **person2**, using the **new keyword followed by the class name**.

### 4.Object.create():

- You can create objects in JavaScript using the **Object.create()** method.
- This method allows you to create an object with a specified prototype.

#### Example:

```
let personPrototype = {  
    greet: function () {  
        console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);  
    },  
}; // Create an object with the specified prototype  
let person = Object.create(personPrototype);  
// Add properties to the object
```



```
person.firstName = "Sangam";
person.lastName = "Kumar";
// Access properties and call the method of the object
console.log(person.firstName);
console.log(person.lastName);
person.greet(); // Output: Hello, my name is Sangam Kumar.
```

#### **Explanation:**

- We create a prototype object called personPrototype that contains a greet method.
- We create a new object called person using Object.create(personPrototype). This means that person inherits the properties and methods of personPrototype.
- We add properties firstName and lastName to the person object.

## **5. Factory Functions:**

- You can create objects in JavaScript using factory functions,
- which are regular functions that return new objects.

#### **Example:**

```
function createPerson(firstName, lastName, age) {
  return {
    firstName: firstName,
    lastName: lastName,
    age: age,
    sayHello: function() {
      console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
    }
  };
}

// Create instances of the person object using the factory function
let person1 = createPerson("Sangam", "Kumar", 15);
let person2 = createPerson("Satyam", "Kumar", 20);

// Access properties and call the method of the objects
console.log(person1.firstName);
console.log(person1.lastName);
console.log(person1.age)
console.log(person2.firstName);
console.log(person2.lastName);
console.log(person2.age);
person1.sayHello(); // Output: Hello, my name is Sangam Kumar.
person2.sayHello(); // Output: Hello, my name is Satyam Kumar.
```

#### **Output:**

```
PS D:\js rev\rajt3> node ./p1.js
Sangam
Kumar
15
Satyam
Kumar
20
Hello, my name is Sangam Kumar.
Hello, my name is Satyam Kumar.
```

#### **Explanation:**

- We define a factory function createPerson that takes firstName, lastName, and age as parameters and returns an object.
- Inside the factory function, we create an object with properties firstName, lastName, and age, along with a sayHello method.
- We create two instances of the person object using the createPerson factory function.

## **❖ 6. Singleton Objects:**

- you can create singleton objects using various patterns.
- One common way is to use an immediately invoked function expression (IIFE) to encapsulate your object creation and ensure that only one instance of the object is created.

#### **Example:**



```
let singleton = (() => {
    // Private variables and functions can be defined here
    let instance;
    // Private initialization logic
    function init() {
        return {
            firstName: "Sangam",
            lastName: "Kumar",
            age: 15,
            sayHello: function() {
                console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
            },
        };
    }
    return {
        // Public method to get the instance of the object
        getInstance: function() {
            if (!instance) {
                instance = init();
            }
            return instance;
        },
    };
})();
```

#### **Creating instances of the singleton object**

```
let obj1 = singleton.getInstance();
let obj2 = singleton.getInstance();
```

#### **Accessing properties and calling the method of the objects**

```
console.log(obj1.firstName); // Output: Sangam
console.log(obj1.lastName); // Output: Kumar
console.log(obj1.age); // Output: 15
console.log(obj2.firstName); // Output: Sangam
console.log(obj2.lastName); // Output: Kumar
console.log(obj2.age); // Output: 15
obj1.sayHello(); // Output: Hello, my name is Sangam Kumar.
obj2.sayHello(); // Output: Hello, my name is Sangam Kumar.
```

#### **Checking if obj1 and obj2 are the same instance**

```
console.log(obj1 === obj2); // Output: true
```

#### **Output:**

```
Sangam
Kumar
15
Sangam
Kumar
15
Hello, my name is Sangam Kumar.
Hello, my name is Sangam Kumar.
true
```



**Explanation:**

- We create a singleton object using an IIFE (immediately invoked function expression) to encapsulate the object creation logic.
- Inside the IIFE, we define a private variable instance to hold the single instance of the object.
- The init function initializes the object with properties and methods.
- We expose a public method getInstance that checks if an instance already exists. If it doesn't, it creates a new instance using the init function and returns it. If an instance already exists, it returns the existing instance.
- We demonstrate obj1 and obj2 are same instance by comparing them with obj1 === obj2.

❖ **7.Using the new Operator with Built-in Constructors:**

- you can create objects using the new operator with built-in constructors like Object, Array, Date, and RegExp.

**Example:**

Creating an Object:// Using the Object constructor to create an empty object

```
let person = new Object(); // Adding properties to the object
```

```
person.firstName = "Sangam";
```

```
person.lastName = "Kumar"; // Accessing properties of the object
```

```
console.log(person.firstName); // Output: Sangam
```

```
console.log(person.lastName); // Output: Kumar
```

Creating an Array:// Using the Array constructor to create an array

```
let numbers = new Array(1, 2, 3, 4, 5); // Accessing elements of the array
```

```
console.log(numbers[0]); // Output: 1
```

```
console.log(numbers[2]); // Output: 3
```

Creating a Date Object:// Using the Date constructor to create a Date object

```
let today = new Date();
```

**Accessing date properties and methods**

```
console.log(today.getFullYear()); // Output: 2023
```

```
console.log(today.getMonth()); // Output: 9
```

Creating a Regular Expression (RegExp):

**Using the RegExp constructor to create a regular expression**

```
let regex = new RegExp("pattern");
```

**Testing the regular expression against a string**

```
let result = regex.test("This is a pattern.");
```

```
console.log(result); // Output: true
```

❖ **JavaScript Properties:**

- Properties are the values associated with a JavaScript object.
- A JavaScript object is a collection of unordered properties.
- Properties can usually be changed, added, and deleted, but some are read only.

❖ **Accessing JavaScript Properties:**

The syntax for accessing the property of an object is:

```
objectName.property // person.age
```

or

```
objectName["property"] // person["age"]
```

or

```
objectName[expression] // x = "age"; person[x]
```

The expression must evaluate to a property name.



## ACCESSING DATA FROM OBJECT

- there are following way you can access data objects:

- 1) Dot Notation
- 2) Bracket Notation
- 3) Computed Property Names (ES6)
- 4) Object Destructuring
- 5) For...In Loop
- 6) Object Methods (e.g., Object.keys(), Object.values(), Object.entries())

### 1. Dot Notation:

#### Example:

#### Creating an object

```
let person = {  
    firstName: "Sangam",  
    lastName: "Kumar",  
    age: 15,  
};
```

#### Output:

```
Sangam  
Kumar  
15
```

#### Accessing data using dot notation

```
console.log(person.firstName);  
console.log(person.lastName);  
console.log(person.age);
```

#### Explanation:

- We create an object named person with properties firstName, lastName, and age.
- To access the data within the object, we use dot notation, where person.firstName accesses the firstName property, person.lastName accesses the lastName property, and person.age accesses the age property.

### 2. Bracket Notation:

- You can access data from an object in JavaScript using bracket notation by specifying the object's name followed by square brackets [ ] and placing the property name you want to access inside the brackets as a string.

#### Example:

```
let person = {  
    firstName: "Sangam",  
    lastName: "Kumar",  
    age: 15,  
};
```

#### Output:

```
Sangam  
Kumar  
15
```

#### // Accessing data using bracket notation

```
console.log(person["firstName"]);  
console.log(person["lastName"]);  
console.log(person["age"]);
```

#### Explanation:

- We create an object named person with properties firstName, lastName, and age.
- To access the data within the object using bracket notation, we place the property name inside square brackets as a string, such as person["firstName"], person["lastName"], and person["age"].

### 3.Computed Property Names (ES6):

// Creating an object with computed property names

```
let propertyName = "firstName";
let person = {
  [propertyName]: "Sangam",
  ["last" + "Name": "Kumar",
  [3 + 3]: 6,
};

console.log(person[propertyName]);
console.log(person["lastName"]);
console.log(person[3 + 3]);
```

**Output:**

```
Sangam
Kumar
6
```

#### Explanation:

- We create an object named person with properties whose names are computed dynamically using computed property names.
- The propertyName variable is used to define the property name as "firstName".
- We also use expressions like ["last" + "Name"] and [2 + 2] to compute property names at runtime.
- When accessing the data, we use square brackets [ ] with the computed property names to retrieve the corresponding values.

### 4.Object Destructuring:

- You can access data from an object in JavaScript using object destructuring.
- Object destructuring allows you to extract specific properties from an object and assign them to variables with the same names as the properties.

**Example://** Creating an object

```
let person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 15,
};

// Using object destructuring to access data
let { firstName, lastName, age } = person;
// Accessing data using the destructured variables
console.log(firstName); // Output: Sangam
console.log(lastName); // Output: Kumar
console.log(age); // Output: 15
```

**Output:**

```
Sangam
Kumar
15
```

#### Explanation:

- We create an object named person with properties firstName, lastName, and age.
- We use object destructuring to extract the properties firstName, lastName, and age from the person object and assign their values to corresponding variables.
- The variables firstName, lastName, and age now hold the values extracted from the object, and we can access them directly.

## 5. For...In Loop:

- You can access data from an object in JavaScript using a for...in loop. This loop iterates through the properties of an object and allows you to access their values.

### Example:

```
// Creating an object
let person = {
  firstName: "Satyam",
  lastName: "Kumar",
  age: 20,
};

// Using a for...in loop to access data
for (let key in person) {
  console.log(` ${key}: ${person[key]}`);
}
```

### Output:

```
firstName: Satyam
lastName: Kumar
age: 20
```

### Explanation:

- We create an object named person with properties firstName, lastName, and age.
- We use a for...in loop to iterate through the properties of the person object.
- Inside the loop, we access the current property's name using the key variable and its corresponding value using person[key].

## 6. Object Methods:

- You can access data from an object in JavaScript using various object methods such as Object.keys(), Object.values(), and Object.entries().
- These methods return arrays containing the keys, values, or key-value pairs (entries) of an object, respectively.
- Using Object.keys():

### Example:

```
// Creating an object
let person = {
  firstName: "Sangam",
  lastName: "Kumar",
  age: 15,
};
// Accessing data using Object.keys()
let keys = Object.keys(person);
// Printing the keys
console.log(keys); // Output: ["firstName", "lastName", "age"]
```

### Output:

```
[ 'firstName', 'lastName', 'age' ]
Using Object.values():
```

## Real Time for Object

```
const prompt = require('prompt-sync')();
// Define district functions
function bihar_dist() {
    console.log("Bihar Districts:");
    console.log("1. Rohtas");
    console.log("2. Patna");
    console.log("3. Betiya");
    console.log("4. ARA");
}
function mp_dist() {
    console.log("Madhya Pradesh Districts:");
    console.log("1. Bhopal");
    console.log("2. Indore");
    console.log("3. Gwalior");
    console.log("4. Jabalpur");
}
// Main state function
function state() {
    const states = {
        "bihar": bihar_dist,
        "madhya pradesh": mp_dist
    };
    let st = prompt("Enter your state name: ").toLowerCase();
    let found = false;
    for (let key in states) {
        if (key === st) {
            states[key](); // Call the respective district function
            found = true;
            break;
        }
    }
    if (!found) {
        console.log("Invalid state. Please try again.");
    }
}
// Call the state function
state();
```

```
Object > JS demo.js > ...
75  function state() {
76      states[key](), // Call the respective district function
77      found = true;
78      break;
79  }
80  if (!found) {
81      console.log("Invalid state. Please try again.");
82  }
83  }
84  // Call the state function
85  state();
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\hp\Desktop\RID ONLINE\object> npm i prompt-sync
added 3 packages in 2s
PS C:\Users\hp\Desktop\RID ONLINE\object> node demo.js
Enter your state name: Bihar
Bihar Districts:
1. Rohtas
2. Patna
3. Betiya
4. ARA
PS C:\Users\hp\Desktop\RID ONLINE\object> []
```



## Adding New Properties in Object

- You can add new properties to an existing object by simply giving it a value.
- Assume that the person object already exists - you can then give it new properties:

### Example-1:

- `person.nationality = "Indian";`

### Example-2:

```
let person={  
    name:"sangam",  
    age: "15",  
    state: "Bihar"  
}  
person.nationality="Indian"  
console.log(person)  
for (let key in person){  
    console.log(key)  
    console.log(key+": "+person[key])  
}
```

Adding new properties

#### Output:

```
{ name: 'sangam', age: '15', state: 'Bihar', nationality:  
'Indian' }  
name  
name:sangam  
age  
age:15  
state  
state:Bihar  
nationality  
nationality:Indian
```

### Example-3:

```
let student={  
  
    // empty object  
}  
// adding new properties and value in student object  
student.name="sangam"  
student.age="15"  
student.city="Patna"  
student.state="Bihar"  
student.Country="Bharat"  
student.Branch="CSE"  
console.log("Object student=",student)  
for(let key in student){  
    console.log(key+": "+student[key])  
}
```

#### Output:

```
Object student={  
    name: 'sangam',  
    age: '15',  
    city: 'Patna',  
    state: 'Bihar',  
    Country: 'Bharat',  
    Branch: 'CSE'  
}  
name: sangam  
age: 15  
city: Patna  
state: Bihar  
Country: Bharat  
Branch: CSE
```

### ❖ How to add multiple key and value at time

```
let money={  
    //Empty Object  
}  
console.log(money)  
const prompt=require('prompt-sync')()  
let n=Number(prompt("Enter the number of key: "))  
for(let i=1; i<=n; i++){  
    keys1=prompt("Enter your key: ")  
    value=prompt("Enter the value: ")  
    money[keys1]=value  
}  
console.log(money)
```

```
PS C:\Users\hp\Desktop\RID ONLINE\function> node obj1.js  
Enter the number of key: 3  
Enter your key: count  
Enter the value: 100  
Enter your key: Note  
Enter the value: yes  
Enter your key: Coin  
Enter the value: No  
{ count: '100', Note: 'yes', Coin: 'No' }  
PS C:\Users\hp\Desktop\RID ONLINE\function>
```



# **Deleting Properties from Object**

- The delete keyword deletes a property from an object:

**Example:**

```
const person = {  
  firstName: "Sangam",  
  lastName: "Kumar",  
  age: 50,  
  eyeColor: "blue"  
};  
console.log("Before delete object=",person)  
delete person.age;  
console.log("After Deleting Properties age",person)
```

**Output:**

```
Before delete object= { firstName: 'Sangam', lastName: 'Kumar', age: 50, eyeColor: 'blue' }  
After Deleting Properties age { firstName: 'Sangam', lastName: 'Kumar', eyeColor: 'blue' }
```

**Note-1:**

- The delete keyword deletes both the value of the property and the property itself.
- After deletion, the property cannot be used before it is added back again.
- The delete operator is designed to be used on object properties. It has no effect on variables or functions.
- The delete operator should not be used on predefined JavaScript object properties. It can crash your application.

**Note-2:**

- To delete only the value of a specific property in an object while keeping the property itself, you can set the **property value to null or undefined**. This way, the key will still exist, but its value will be cleared.

**Example:**

```
const person = {  
  firstName: "Sangam",  
  lastName: "Kumar",  
  age: 50,  
  eyeColor: "blue"  
};  
console.log("Before deleting value of lastName:", person);  
person["lastName"] = null; // Setting the value of lastName to null  
console.log("After deleting value of lastName:", person);  
delete person["age"]; // Deleting the age property  
console.log("After deleting property age:", person);  
person.lastName="Prasad" // adding lastName value is prasad  
console.log("After adding lastNmae value:", person);
```

**Output:**

```
Before deleting value of lastName: { firstName: 'Sangam', lastName: 'Kumar', age: 50, eyeColor: 'blue' }  
After deleting value of lastName: { firstName: 'Sangam', lastName: null, age: 50, eyeColor: 'blue' }  
After deleting property age: { firstName: 'Sangam', lastName: null, eyeColor: 'blue' }  
After adding lastNmae value: { firstName: 'Sangam', lastName: 'Prasad', eyeColor: 'blue' }
```

**Or:**

```
const person = {  
  firstName: "Sangam",  
  lastName: "Kumar",  
  age: 50,  
  eyeColor: "blue"  
};  
console.log("Before delete object=",person)  
delete person["age"];  
console.log("After Deleting Properties age",person)
```



## ❖ Nested Objects:

- Nested objects in JavaScript are objects within other objects. They allow you to create complex data structures that can represent real-world entities more accurately.

### Example-1 :

```
const student={  
marks: {  
    maths: 100,  
    Science: 98,  
    English: 56,  
    Hindi: 78  
},  
food_name: ["panipuri","momos","Burger"],  
College_name: "Oxford Business College",  
Birthday: {  
    party: "yes",  
    Gift: "pen",  
    Cake: "3kg"  
},  
age: 25  
}  
console.log(marks.English)
```

```
for (let i in student){  
    console.log(i)  
} console.log(student["Birthday"]["Gift"]) // gift is key of nested object  
    console.log(student["food_name"][0]) // 0 is a index of array  
// console.log(student["food_name"])
```

```
b=student.food_name  
for (let i of b){  
    console.log(i)  
}  
for (let j in b){  
    console.log(j)  
    console.log(j,":",b[j])  
}
```

```
PS C:\Users\hp\Desktop\RID ONLINE\function> node obj2.js  
pen  
panipuri  
0  
1  
2  
PS C:\Users\hp\Desktop\RID ONLINE\function> node obj2.js  
pen  
panipuri  
0 : [ '0' ]  
1 : [ '1' ]  
2 : [ '2' ]  
PS C:\Users\hp\Desktop\RID ONLINE\function> node obj2.js  
pen  
panipuri  
0 : panipuri  
1 : momos  
2 : Burger  
PS C:\Users\hp\Desktop\RID ONLINE\function>
```

### Example-2:

```
let person={  
name:"Sangam",  
age:25,  
fav:["red","blue","green"],  
marks:{  
    maths:50,  
    sci:70,  
    hindi:90  
}  
}
```

```
console.log(person)  
console.log(person["name"])//Access value in object via key- method 1  
console.log(person.name)//Access value in object via key- method 2  
console.log(person.fav[2],person.fav[0],person.fav[1])  
console.log(person.marks.sci)  
console.log(person["marks"]["sci"])  
//Nested Objects, Values in an object be in another object
```

### Output:

```
{  
    name: 'Sangam',  
    age: 25,  
    fav: [ 'red', 'blue', 'green' ],  
    marks: { maths: 50, sci: 70, hindi: 90 }  
}  
Sangam  
Sangam  
green red blue  
70  
70
```

**Example-3:**

```
const person = {
    name: "Sangam Kumar",
    age: 15,
    address: {
        street: "123 Main St",
        city: "Patna",
        country: "Bihar"
    };
    console.log(person.name);
    console.log(person.address.street);
    console.log(person.address.city);
```

**Output:** Sangam Kumar  
123 Main St  
Patna

**Example-4**

```
const book = {
    title: "JavaScript Basics",
    pages: 250,
    author: {
        firstName: "Jane",
        lastName: "Smith",
        nationality: "Canadian"
    };
// Accessing nested object properties
console.log(book.title); // Output: JavaScript Basics
console.log(book.author.firstName); // Output: Jane
console.log(book.author.nationality); // Output: Canadian
```

## ❖ Nested Arrays and Objects:

- Values in objects can be arrays, and values in arrays can be objects:

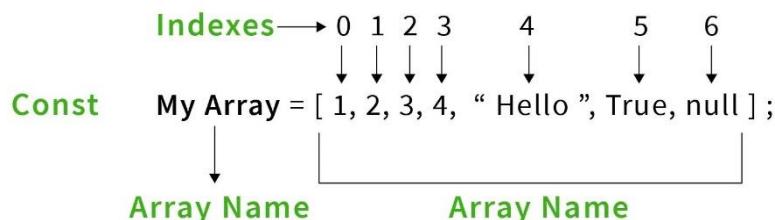
**Example:**

```
const school = {
    name: "Delhi Public School",
    address: {
        street: "45 MG Road", city: "Delhi", zip: "110001"
    },
    classes: [ {
        name: "Math 101",
        teacher: "Mr. Sharma",
        students: [
            { name: "Aarav", age: 14, grade: "A" },
            { name: "Ananya", age: 15, grade: "B" }
        ]
    },
    {
        name: "Science 102",
        teacher: "Ms. Mehta",
        students: [
            { name: "Vivaan", age: 14, grade: "A" },
            { name: "Isha", age: 15, grade: "B+" }
        ]
    }
]; // Accessing nested values
console.log("School Name:", school.name); // Delhi Public School
console.log("City:", school.address.city); // Delhi
console.log("First Class Teacher:", school.classes[0].teacher); // Mr. Sharma
console.log("Second Student Name in Math 101:", school.classes[0].students[1].name);
school.classes[1].students.push({ name: "Aryan", age: 16, grade: "A-" });
console.log("Updated Students in Science 102:", school.classes[1].students);
school.classes.forEach((classObj) => {
    console.log(`Class: ${classObj.name}, Teacher: ${classObj.teacher}`);
    classObj.students.forEach((student) => {
        console.log(` - Student: ${student.name}, Grade: ${student.grade}`);
    });
});
```



# ARRAY

- Array is a non-primitive data type that stores multiple values in a single variable.
- Arrays can hold elements of different data types, including numbers, strings, objects, functions, and other arrays.
- arrays are indexed, ordered, and mutable, meaning you can add, remove, and modify elements.



## ❖ Creating Arrays:

- You can create an array in JavaScript using in this following way.
  1. **Array Literals**
  2. **Array Constructor**
  3. **Array.of**
  4. **Array.from**

### 1. Using Array Literals

- **Description:** The most straightforward and commonly used way to create arrays. You define the array directly using square brackets.
- **Syntax:** `const arrayName = [element1, element2, ...]`
- **Example:**

```

const fruits = ["apple", "banana", "cherry"];
const array1=["mohan", "sohan",10,20,3.6,{name: "Sangam"},true]
const a=[3,4,23,23,23,242,44,242,4,234523]
console.log("type of fruits=",typeof(fruits))
console.log("type of array1=",typeof(array1))
console.log("type of a=",typeof(a))
  
```

#### Output:

```

type of fruits=object
type of array1=object
type of a=object
  
```

### 2. Using the Array Constructor

- **Description:** You can use the Array constructor to create arrays. Be cautious when using a single number as an argument—it will create an array with that number of empty slots.
- **Syntax:**

```

const arrayName = new Array(element1, element2, ...);
const arrayName = new Array(arrayLength);
  
```

#### Example-2:

```

const b=new Array(1)
console.log(b)
  
```

#### Output:

```

PS C:\Users\hp > node demo.js
[ <1 empty item> ]
  
```

- **Example-1:**

```

const cars = new Array("Toyota", "Honda", "Ford");
const data=new Array(12,24,4,232,32,324,23)
const v= new Array ("mohan", "sohan",10,20,3.6,{name: "Sangam"},true)
const name=new Array("sangam", "Sushil", "Sujeet", "Satyam", "priti", "Roshni")
console.log("type of data=",typeof(data))
console.log("type of v=",typeof(v))
console.log("type of name=",typeof(v))
  
```

**Output:** type of data= object  
type of v= object  
type of name= object

- **Example-3**

```

const numbers = new Array(1, 2, 3); // Creates [1, 2, 3]
console.log(numbers);
const emptySlots = new Array(5); // Creates an array with
5 empty slots
console.log(emptySlots); // Output: [ <5 empty items> ]
  
```

### 3. Using Array.of

- **Description:** Creates an array from the arguments provided, including single numbers (avoids ambiguity with the Array constructor).
- **Syntax:** `const arrayName = Array.of(element1, element2, ...);`
- **Example:**

```
const mixedArray = Array.of(5, 'hello', true);
console.log(mixedArray); // Output: [5, 'hello', true]
```

### 4. Using Array.from

- **Description:** Creates an array from an iterable (like a string, map, set, or array-like objects such as arguments).
- **Syntax:** `const arrayName = Array.from(iterable[, mapFunction[, thisArg]]);`
- **Example:** with a string

```
const charArray = Array.from('123');
console.log(charArray); // Output: ['1', '2', '3']
```
- **Example with a map function**

```
const squares = Array.from([1, 2, 3], x => x ** 2);
console.log(squares); // Output: [1, 4, 9]
```

**Note:** If you intend to convert an object into an array of its values, you would need to use a different approach like `Object.entries()` or `Object.values()`.

#### Example:

##### ➤ Incorrect usage of Array.from() with an object

```
const a = Array.from({ name: "raj", roll_no: 53 });
console.log(a);
```

**Output:** [] - Array.from doesn't work with plain objects

##### ➤ Convert object to an array of key-value pairs

```
const b = Array.from(Object.entries({ name: "raj", roll_no: 53 }));
console.log(b);
```

**Output:** [["name", "raj"], ["roll\_no", 53]]

##### ➤ Convert object to an array of values

```
const c = Array.from(Object.values({ name: "raj", roll_no: 53 }));
console.log(c);
```

**Output:** ["raj", 53]

##### ➤ Convert object to an array of keys

```
const e = Array.from(Object.keys({ name: "raj", roll_no: 53 }));
console.log(e);
```

**Output:** ["name", "roll\_no"]

## ❖ How to access data from the array

- This following ways to access data from an array

1. Using Index
2. Using Loops (for, for...of)
3. Using Array.find
4. Using Array.filter

### 1. Using Index

- **Description:** You can access array elements using square brackets[] and index value Access elements directly by their index. index starts from 0.

- **Syntax:** arrayName[index];

- **Example-1:**

```
const arr = [10, 20, 30];
console.log(arr[0]); // Output: 10
console.log(arr[2]); // Output: 30
```

- **Example-2:** const fruits = ["apple", "banana", "cherry"]

```
console.info("vlaue of array fruits:")
console.log(fruits[0]) console.log(fruits[1])
const v= new Array ("mohan", "sohan",10,20,3.6,{name: "Sangam"},true)
console.info("vlaue of array v:")
console.log(v[0]) console.log(v[3])
const array1=["mohan", "sohan",10,20,3.6,{name: "Sangam"},true]
console.info("vlaue of array array1:") console.log(array1[2]) console.log(array1[5])
```

#### Output:

```
vlaue of array fruits:
apple
banana
vlaue of array v:
mohan
20
vlaue of array array1:
10
```

### 2. Using Loops (for, for...of)

- **For Loop:** **Description:** Iterate over the array using the index.

- **Syntax:**

```
for (let i = 0; i < arrayName.length; i++) {
    // Access arrayName[i]
}
```

- **Example-1:**

```
const arr = [10, 20, 30];
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]); // Output: 10, 20, 30
}
```

- **Example-2:**

```
const a=[10,20,30]
console.log(a[0])
for(let i=0;i<a.length;i++){
    console.log(i,":",a[i])
}
```

**Output:** 10

0 : 10

1 : 20

2 : 30

- You can use loop for iterate or access the elements from an array.
- There are following loops can used for access the elements from an array
  - 1. for loop
  - 2. forEach
  - 3. for...of
  - 4. while loop and others..



## ➤ For...of Loop

- **Description:** Directly iterate over the elements of the array.
- **Syntax:**

```
for (const item of arrayName) {
    // Access item
}
```

### Example-1:

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

### Example-2:

```
const a=[10,20,30]
console.log("....Result-1...")
for(let i=0;i<a.length;i++){
    console.log(i,":",a[i])
}
console.log("....Result-2...")
for(let i in a){
    console.log(i,":",a[i])
}
```

#### Output:

....Result-1...

0 : 10

1 : 20

2 : 30

....Result-2...

0 : 10

1 : 20

2 : 30

### Example-3:

```
const a=[10,20,30,40,50]
a.forEach(function(a){
    console.log(a)
})
```

#### Output:

10

20

30

### Example-4:-by using the fof loop.

```
const fruits = ["apple", "banana", "cherry"];
```

#### Using a for loop

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

#### Using forEach

```
fruits.forEach(function (fruit) {
    console.log(fruit);
});
```

#### Using for...of

```
for (const fruit of fruits) {
    console.log(fruit);
}
```

#### Output:

10

20

30

### Example-5

```
const a = [10, 20, 30];
let i = 0;
while (i < a.length) {
    console.log(i, ":", a[i]);
    i++;
}
```

#### Output:

10

20

30

### Example-6:

```
const a = [10, 20, 30];
let i = 0;
do {
    console.log(i, ":", a[i]);
    i++;
} while (i < a.length);
```

## 3. Using Array.find

- **Description:** Returns the first element that matches a condition (predicate function). If no match is found, it returns undefined.
- **Syntax:** arrayName.find(callback(element, index, array));
- **Example:**

```
const arr = [10, 20, 30];
const result = arr.find(x => x > 15);
console.log(result); // Output: 20
```

## 4. Using Array.filter

- **Description:** Returns a new array containing all elements that match a condition (predicate function).
- **Syntax:** arrayName.filter(callback(element, index, array));
- **Example:** const arr = [10, 20, 30];
 const result = arr.filter(x => x > 15);
 console.log(result); // Output: [20, 30]

#### Example:

```
const a = [10, 20, 30, 40, 50];
const b = a.filter(function(x) {
    return x > 15; // Condition to
    filter elements greater than 15
});
console.log(b);
```



### 3. Modifying Arrays:

- Arrays in JavaScript are mutable, which means you can change their content by adding, removing, or updating elements:

#### 1) Adding elements in array by using the push() and unshift() method

- Push():-** method is used for add the elements in end of the array array
- unshift():-** method is used for add the elements in beginning of the array array
- [index]=value:-** using the index value
- splice() :-** To add multiple elements at a **specific position** (index) in an array, you can use the splice() method. It allows you to **insert** elements at any position.

**Note:** If you want to add multiple elements at a **specific position**, you cannot directly use push() or unshift().you can use these methods along with splice() to insert elements at a specific position.

##### A. push() Method

- Description:** Adds one or more elements to the end of the array and returns the new length of the array.
- Syntax:** arrayName.push(element1, element2, ...);
- Example:**

```
const arr = [10, 20, 30];
arr.push(40, 50);
console.log(arr); // Output: [10, 20, 30, 40, 50]
```

##### B. unshift() Method

- Description:** Adds one or more elements to the beginning of the array and returns the new length of the array.
- Syntax:** arrayName.unshift(element1, element2, ...);
- Example:** const arr = [10, 20, 30];
arr.unshift(0, 5);
console.log(arr); // Output: [0, 5, 10, 20, 30]

**Example:** const fruits = ["apple", "banana", "cherry"];
console.info("Before adding element in array")
console.log("array=",fruits)
fruits.push("orange"); // Adds "orange" to the end
fruits.unshift("grape"); // Adds "grape" to the beginning
console.info("after adding element in array")
console.log("array=",fruits)

**Output:**
Before adding element in array
array= [ 'apple', 'banana', 'cherry' ]
after adding element in array
array= [ 'grape', 'apple', 'banana', 'cherry',
'orange' ]
console.log("array=",fruits)

#### C. Updating elements in specific index.

- By using the index value we can add the element on specific position in array

**Syntax:** arrayName[index] = newValue;

##### Example:

```
const arr = [10, 20, 30, 40];
// Update element at index 2 (change 30 to 35)
arr[2] = 35;
console.log(arr); // Output: [10, 20, 35, 40]
```

### Note-1: Updating an Existing Element in the Array

- If the index is within the current length of the array, the element at that index will be updated.

**Example:** const arr = [10, 20, 30]; // Update the element at index 1  
arr[1] = 25;  
console.log(arr); // Output: [10, 25, 30]

### Note- 2. Adding an Element at a New Index (Out of Bounds)

- If index is greater than current length, the array will be extended with **empty slots**.

**Example:** const arr = [10, 20, 30]; // Try to add value at index 5 (which doesn't exist yet)  
arr[5] = 40;  
console.log(arr); // Output: [10, 20, 30, <2 empty items>, 40]  
console.log(arr.length); // Output: 6 (new length)

- Here, Js extends array up to index 5, adding **empty slots** (<2 empty items>) and placing 40 at index 5.

### Note-3: Adding Multiple Elements (Indirectly)

- If you directly assign a value to an index higher than the current length, it doesn't automatically add multiple elements. The missing slots will remain empty, but the value will be placed at the specified index.

#### Example:

```
const arr = [1, 2]; // Assign a value at index 4 (index 0, 1, 2, 3 will remain empty)  
arr[4] = 10;  
console.log(arr); // Output: [1, 2, <2 empty items>, 10]
```

#### ❖ Important Points:

- Array will grow:** If you assign a value to an index that is larger than the current length, the array will grow, filling the intermediate positions with **empty slots**.
- Sparse Arrays:** The result will be a **sparse array**, which contains **empty slots** (<empty>) at positions where no values were explicitly added.
- Direct assignment** doesn't automatically push or add new elements at the end. It only places the value at the specific index, and missing positions are filled with **empty slots**.

## D.Adding Elements at a Specific Index Using splice()

- To add multiple elements at a **specific position** (index) in an array, you can use the **splice()** method. It allows you to **insert** elements at any position.
- Syntax for splice():** `arrayName.splice(index, 0, element1, element2, ...);`
  - ✓ **index:** The position at which to insert elements (starting from 0).
  - ✓ **0:** The number of elements to remove (we set it to 0 to prevent removal).
  - ✓ **element1, element2, ...:** The elements to add at the specific position.
- Syntax:** `array.splice(index, deleteCount, element1, element2, ...);`
  - ✓ **index** is where you want to insert the elements.
  - ✓ **deleteCount** is how many elements you want to remove (set to 0 if you don't want to remove any).
  - ✓ **elements** after the deleteCount are the ones to be inserted.

**Example:**

```
const arr = [10, 20, 30, 40]; // Add elements at index 2 (before 30)
arr.splice(2, 0, 25, 27);
console.log(arr); // Output: [10, 20, 25, 27, 30, 40]
```

**1. Adding Multiple Values at the Beginning of the Array**

- You can use splice() to add multiple values at the **start** of the array by specifying an index of 0.

**Example:**

```
const arr = [30, 40, 50]; // Add multiple elements at index 0 (beginning of the array)
arr.splice(0, 0, 10, 20);
console.log(arr); // Output: [10, 20, 30, 40, 50]
```

**splice(0, 0, 10, 20):**

- ✓ 0 indicates the index to insert.
- ✓ The second 0 means we do not remove any elements.
- ✓ 10 and 20 are the elements being added at the beginning.

**2. Adding Multiple Values in the Middle of the Array**

- You can add multiple values at any position in the middle of the array by specifying the correct index.

**Example:**

```
const arr = [10, 20, 30, 40, 50]; // Add multiple elements at index 2
arr.splice(2, 0, 25, 27);
console.log(arr); // Output: [10, 20, 25, 27, 30, 40, 50]
```

**splice(2, 0, 25, 27):**

- ✓ 2 is the index where the new elements (25 and 27) will be inserted.
- ✓ The second 0 indicates no elements will be removed.

**3. Adding Multiple Values at the End of the Array**

- You can also use splice() to add multiple elements at the end of the array by using the last index.

**Example:**

```
const arr = [10, 20, 30]; // Add multiple elements at the end (after index 2)
arr.splice(arr.length, 0, 40, 50, 60);
console.log(arr); // Output: [10, 20, 30, 40, 50, 60]
```

**splice(arr.length, 0, 40, 50, 60):**

- ✓ arr.length gives the current length of the array, which represents the next available index for insertion.
- ✓ The second 0 indicates no elements are being removed.
- ✓ 40, 50, 60 are the values being added at the end of the array.

## 2) Removing elements from array by using pop() and shift() method.

- A. **Pop():-** method is used for remove the elements in end of the array.
- B. **shift():-** method is used for remove the elements in beginning of the array.
- C. **splice():-** Removes elements at a specific position and can remove multiple elements.
- D. **Delete:-** Deletes an element at a specific index, leaving an empty slot (undefined).

### A. pop() Method:

- **Description:** Removes the last element from the array and returns it. This method modifies the original array.
- **Note:** Removes one element at a time. Use a loop or splice() for multiple removals.
- **Syntax:** arrayName.pop();
- **Example:** const arr = [10, 20, 30, 40];  
const removedElement = arr.pop();  
console.log(arr); // Output: [10, 20, 30]  
console.log(removedElement); // Output: 40

### B. shift() Method

- **Description:** Removes the first element from the array and returns it. This method modifies the original array.
- **Note:** it can only remove **one element at a time** from the beginning of the array.
- **Syntax:** arrayName.shift();
- **Example:** const arr = [10, 20, 30, 40];  
const removedElement = arr.shift();  
console.log(arr); // Output: [20, 30, 40]  
console.log(removedElement); // Output: 10

#### Example:

```
const fruits = ["apple", "banana", "cherry"];
console.info("Before remove element in array")
console.log("array=",fruits)
fruits.pop("orange"); // remove "orange" to the end
fruits.shift("grape"); // remove "grape" to the beginning
console.info("after remove element in array")
console.log("array=",fruits)
```

#### Output:

```
Before remove element in array
array= [ 'apple', 'banana', 'cherry' ]
after remove element in array
array= [ 'banana' ]
```

### C. splice()

- **Description:** Removes elements at a specific position and can remove multiple elements.
- **Syntax:** array.splice(index, deleteCount);
  - ✓ **index:** The position where removal starts.
  - ✓ **deleteCount:** The number of elements to remove.
- **Example-1:**

```
const arr = [10, 20, 30, 40];
arr.splice(2, 1); // Removes the element at index 2
console.log(arr); // Output: [10, 20, 40]
```

#### Example -2: Removing Multiple Consecutive Elements

```
const arr = [10, 20, 30, 40, 50, 60]; // Remove 3 elements starting from index 2 (30, 40, 50)
arr.splice(2, 3);
console.log(arr); // Output: [10, 20, 60]
```

#### splice(2, 3):

- ✓ 2 is the index where the removal starts.
- ✓ 3 is the number of elements to remove (30, 40, 50).



### Example-3: Removing Multiple Elements from the End

```
const arr = [100, 200, 300, 400, 500]; // Remove 2 elements starting from index 3 (400, 500)
arr.splice(3, 2);
console.log(arr); // Output: [100, 200, 300]
```

#### splice(3, 2):

- ✓ 3 is the index to start removing from (starting at 400).
- ✓ 2 is the number of elements to remove (400, 500).

### Example-4: Removing More Elements Than Exist in the Array

```
const arr = [10, 20, 30];
// Try to remove 5 elements starting from index 1 (only 2 elements left after index 1)
arr.splice(1, 5);
console.log(arr); // Output: [10]
```

#### splice(1, 5):

- ✓ 1 is the index where removal starts (from 20).
- ✓ 5 specifies that we want to remove 5 elements, but there are only 2 elements left (20, 30). So, only those 2 elements will be removed.

**D.delete:-** Deletes an element at a specific index, leaving an empty slot (undefined).

- **Syntax:** delete array[index];
- **Example:**

```
const arr = [10, 20, 30, 40];
delete arr[2]; // Removes the element at index 2
console.log(arr); // Output: [10, 20, <1 empty item>, 40]
```

### Difference Between `delete` and `splice()`

Feature	<code>delete</code>	<code>splice()</code>
Purpose	Removes an element at a specific index from an array.	Removes or adds elements at a specific index in an array.
Effect on Array	Leaves an empty slot ( <code>undefined</code> ) at the removed index.	Removes elements and shifts the rest of the array to fill the gap.
Return Value	<code>undefined</code>	The removed elements as an array.
Array Length	Does not change the array's length; only creates empty slots.	Changes the array's length by removing or adding elements.
Usage	Primarily for removing an element without affecting other elements.	Primarily for modifying the array by removing or adding elements.

### Example 1: Using `delete`

- The `delete` operator removes an element from a specified index but **does not adjust the array**. It leaves an empty slot (`undefined`) at the position of the removed element.

#### Example:

```
const arr = [10, 20, 30, 40];
delete arr[2]; // Removes the element at index 2 (30)
console.log(arr); // Output: [10, 20, <1 empty item>, 40]
console.log(arr.length); // Output: 4
```

- The element at index 2 is removed, but an empty slot (`<1 empty item>`) is left in the array.
- The array's length remains the same (4), but the element is **not shifted**.



### Example 2: Using splice()

- The splice() method removes elements at a specific index and shifts the remaining elements to fill the gap. It also modifies the array's length.

**Example:**

```
const arr = [10, 20, 30, 40];
arr.splice(2, 1); // Removes the element at index 2 (30)
console.log(arr); // Output: [10, 20, 40]
console.log(arr.length); // Output: 3
```

- splice(2, 1) removes the element 30 from index 2 and shifts remaining elements to the left.
- The array's length is now 3 because the element has been removed, and the remaining elements have been shifted.

**4. Array Length:** You can find the number of elements in an array using the `length` property:

- Example:**

```
const fruits = ["apple", "banana", "cherry"];
console.log("length of array=", fruits.length);
console.info("length of array=", fruits.length)
const a=new Array(10,4,2,3,5, true, "skills")
console.log("length of array=", a.length)
console.warn("length of array=", a.length)
console.error("length of array=", a.length)
```

**Output:**

```
length of array= 3
length of array= 3
length of array= 6
length of array= 6
length of array= 6
length of array= 6
```

## ARRAY METHOD IN JS

### ❖ Mutating Methods (Modify the original array):

- push()** – Adds elements to the end of the array.
- pop()** – Removes the last element from the array.
- shift()** – Removes the first element from the array.
- unshift()** – Adds elements to the beginning of the array.
- splice()** – Adds, removes, or replaces elements at a specific index.
- reverse()** – Reverses the order of the array.
- sort()** – Sorts the array in place.
- copyWithin()** – Copies part of the array to another location within the same array.
- fill()** – Fills the array with a static value.

### ❖ Non-Mutating Methods (Return a new array or value without modifying original array)

- concat()** – Merges two or more arrays.
- slice()** – Returns a shallow copy of a portion of an array.
- map()** – Creates a new array by applying a function to each element.
- filter()** – Creates a new array with elements that satisfy a condition.
- reduce()** – Reduces the array to a single value by applying a function.
- reduceRight()** – Like reduce(), but processes the array from right to left.
- flat()** – Flattens nested arrays into a single array.
- flatMap()** – Combines mapping and flattening in one step.
- join()** – Joins all elements into a string with a specified separator.

### ❖ Iterative Methods (Perform actions on each element)

- forEach()** – Executes a function for each array element.
- every()** – Returns true if all elements satisfy a condition.
- some()** – Returns true if at least one element satisfies a condition.



4. **find()** – Returns the first element that satisfies a condition.
5. **findIndex()** – Returns the index of the first element that satisfies a condition.
6. **findLast()** – Returns the last element that satisfies a condition (ES2023+).
7. **findLastIndex()** – Returns the index of the last element that satisfies a condition (ES2023+).

### ❖ Access and Information Methods

1. **includes()** – Checks if an array contains a specific value.
2. **indexOf()** – Returns the first index of a value.
3. **lastIndexOf()** – Returns the last index of a value.
4. **keys()** – Returns an iterator for the array's keys (indices).
5. **values()** – Returns an iterator for the array's values.
6. **entries()** – Returns an iterator for key-value pairs.
7. **length** – Property to get the size of the array.

### ❖ Type Conversion

1. **toString()** – Converts an array to a comma-separated string.
2. **toLocaleString()** – Converts elements to strings using their locale-specific representation.

### ❖ Array Creation

1. **Array.from()** – Creates an array from an iterable or array-like object.
2. **Array.of()** – Creates an array from a set of arguments.

## Mutating Methods (Modify the original array)

### 1. **push()**

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

**Syntax:** array.push(element1, element2, ...)

**Example:**

```
let fruits = ["apple", "banana"];
fruits.push("cherry");
console.log(fruits);
```

**Output:** ["apple", "banana", "cherry"]

### 2. **pop()**

**Description:** Removes last element from array and returns it.

**Syntax:** array.pop()

**Example:**

```
let fruits = ["apple", "banana", "cherry"];
let removed = fruits.pop();
console.log(fruits); // ["apple", "banana"]
```

### 3. **shift()**

**Description:** Removes the first element from the array and returns it.

**Syntax:** array.shift()

**Example:**

```
let fruits = ["apple", "banana", "cherry"];
let removed = fruits.shift();
console.log(fruits); // ["banana", "cherry"]
console.log(removed); // "apple"
```

### 4. **unshift()**

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

**Syntax:** array.unshift(element1, element2, ...)

**Example:**

```
let fruits = ["banana", "cherry"];
fruits.unshift("apple");
console.log(fruits); // ["apple", "banana", "cherry"]
```

### 5. **splice()**

**Description:** Adds, removes, or replaces elements at a specific index.

**Syntax:** array.splice(start, deleteCount, item1, item2, ...)

**Example:**

```
let fruits = ["apple", "banana", "cherry"];
fruits.splice(1, 1, "orange");
console.log(fruits); // ["apple", "orange", "cherry"]
```



## 6. reverse()

**Description:** Reverses order of the array.

**Syntax:** array.reverse()

**Example:**

```
let numbers = [1, 2, 3];
numbers.reverse();
console.log(numbers); // [3, 2, 1]
```

## 8. copyWithin()

**Description:** Copies a sequence of array elements to another location within the same array.

**Syntax:** array.copyWithin(target, start, end)

**Example:**

```
let numbers = [1, 2, 3, 4, 5];
numbers.copyWithin(1, 3, 5);
console.log(numbers);
```

**Output:** [1, 4, 5, 4, 5]

## 7. sort()

**Description:** Sorts the elements of the array in place. By default, it sorts elements as strings.

**Syntax:** array.sort(compareFunction)

**Example:**

```
let numbers = [3, 1, 4, 1];
numbers.sort();
console.log(numbers); // [1, 1, 3, 4] (default lexicographical order)
```

For numerical order:

## 9. fill()

**Description:** Fills elements in the array with a static value.

**Syntax:** array.fill(value, start, end)

**Example:**

```
let numbers = [1, 2, 3, 4];
numbers.fill(0, 1, 3);
console.log(numbers); // [1, 0, 0, 4]
```

# Non-Mutating Methods in JavaScript

- Non-mutating methods are array methods in JavaScript that do not alter the original array. Instead, they return a new array or a specific value, ensuring the immutability of the original array.

## 1. concat():-

Merges two or more arrays into a new array.

**Syntax:** array1.concat(array2, array3, ...)

**Example:**

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const result = arr1.concat(arr2);
console.log(result); // [1, 2, 3, 4]
console.log(arr1);
Output: [1, 2] (original array remains unchanged)
```

**2. slice():** Returns a shallow copy of a portion of an array, specified by the start and end indices.

**Syntax:** array.slice(start, end)

**Example:** const arr = [10, 20, 30, 40];

```
const sliced = arr.slice(1, 3);
```

```
console.log(sliced); // [20, 30]
```

```
console.log(arr);
```

**Output:** [10, 20, 30, 40] (original array remains unchanged)

## 3. map():

Creates a new array by applying a given function to each element of the original array.

**Syntax:** array.map(callback(element, index, array))

**Example:**

```
const numbers = [1, 2, 3];
const squared = numbers.map(num => num * num);
console.log(squared); // [1, 4, 9]
console.log(numbers); // [1, 2, 3]
```

**4. filter():** Creates a new array with elements that satisfy a specified condition.

**Syntax:** array.filter(callback(element, index, array))

**Example:** const numbers = [1, 2, 3, 4];

```
const even = numbers.filter(num => num % 2
    === 0);
```

```
console.log(even); // [2, 4]
```

```
console.log(numbers); // [1, 2, 3, 4]
```

## 5. reduce()

**Description:** Reduces array to a single value by applying a function to each element, from left to right.

**Syntax:** array.reduce(callback(accumulator, currentValue, index, array), initialValue)

**Example:** const numbers = [1, 2, 3, 4];

```
const sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // 10
```



**6. reduceRight():** Similar to reduce(), but processes the array from right to left.

**Syntax:** array.reduceRight(callback(accumulator, currentValue, index, array), initialValue)

**Example:**

```
const numbers = [1, 2, 3, 4];
const product = numbers.reduceRight((acc, num) => acc * num, 1);
console.log(product); // 24
```

**7. flat():** Flattens nested arrays into a single array, up to a specified depth.

**Syntax:** array.flat(depth)

**Example:**

```
const nested = [1, [2, [3, [4]]]];
const flattened = nested.flat(2);
console.log(flattened); // [1, 2, 3, [4]]
```

**8. flatMap():** Combines mapping and flattening of arrays in one step. Equivalent to performing map() followed by flat(1).

**Syntax:** array.flatMap(callback(element, index, array))

**Example:**

```
const numbers = [1, 2, 3];
const result = numbers.flatMap(num => [num, num * 2]);
console.log(result); // [1, 2, 2, 4, 3, 6]
```

**9. join():** Joins all elements of an array into a string, separated by a specified delimiter.

**Syntax:** array.join(separator)

**Example:**

```
const words = ['Hello', 'world'];
const sentence = words.join(' ');
console.log(sentence); // "Hello world"
```

**Note:** These methods are essential tools for working with arrays in JavaScript, providing versatile ways to manipulate and analyze data while maintaining the integrity of the original array.

## Iterative Methods in JavaScript

- Iterative methods are array methods that perform specific actions on each element of the array. They help in iterating over arrays and performing tasks like checking conditions, finding elements, or executing functions.

**1. forEach():** Executes a provided function for each element in the array.

**Syntax:** array.forEach(callback(element, index, array))

**Example:**

```
const numbers = [1, 2, 3];
numbers.forEach(num => console.log(num));
```

**Output:** 1, 2, 3

**2. every():** Returns true if all elements in array satisfy a specified condition; otherwise, returns false.

**Syntax:** array.every(callback(element, index, array))

**Example:**

```
const numbers = [2, 4, 6];
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // true
```

**3. some():** Returns true if at least one element in the array satisfies a specified condition; otherwise, returns false.

**Syntax:** array.some(callback(element, index, array))

**Example:**

```
const numbers = [1, 3, 5, 6];
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // true
```

**4. find():** Returns the first element in the array that satisfies a specified condition. If no such element is found, returns undefined.

**Syntax:** array.find(callback(element, index, array))

**Example:**

```
const numbers = [5, 10, 15];
const greaterThanTen = numbers.find(num => num > 10);
console.log(greaterThanTen); // 15
```

**5. findIndex():** Returns the index of the first element that satisfies a specified condition. If no such element is found, returns -1.

**Syntax:** array.findIndex(callback(element, index, array))

**Example:**

```
const numbers = [5, 10, 15];
const index = numbers.findIndex(num => num > 10);
console.log(index); // 2
```

**6. findLast() (ES2023+):** Returns the last element in the array that satisfies a specified condition. If no such element is found, returns undefined.

**Syntax:** array.findLast(callback(element, index, array))

**Example:**

```
const numbers = [10, 20, 30];
const lastGreaterThan15 = numbers.findLast(num => num > 15);
console.log(lastGreaterThan15); // 30
```

**7. findLastIndex() (ES2023+):** Returns the index of the last element that satisfies a specified condition. If no such element is found, returns -1.

**Syntax:** array.findLastIndex(callback(element, index, array))

**Example:**

```
const numbers = [10, 20, 30];
const lastIndex = numbers.findLastIndex(num => num > 15);
console.log(lastIndex); // 2
```

## Access and Information Methods in JavaScript

- Access and information methods in JavaScript provide ways to retrieve values, indices, or size-related details of an array. These methods are useful for checking array contents and iterating over elements.

**1. includes():-** Checks if array contains specific value. Returns true if the value exists, otherwise false.

**Syntax:** array.includes(value, fromIndex)

**Example:**

```
const fruits = ['apple', 'banana', 'cherry'];
console.log(fruits.includes('banana')); // true
console.log(fruits.includes('grape')); // false
```

**2. indexOf():-** Returns first index of a specified value in the array. Returns -1 if the value is not found.

**Syntax:** array.indexOf(value, fromIndex)

**Example:**



```
const numbers = [10, 20, 30, 10];
console.log(numbers.indexOf(10)); // 0
console.log(numbers.indexOf(40)); // -1
```

**3. lastIndexOf():** Returns last index of a specified value in array. Returns -1 if the value is not found.

**Syntax:** array.lastIndexOf(value, fromIndex)

**Example:**

```
const numbers = [10, 20, 30, 10];
console.log(numbers.lastIndexOf(10)); // 3
```

**4. keys():** Returns an iterator containing the keys (indices) of the array.

**Syntax:** array.keys()

**Example:**

```
const fruits = ['apple', 'banana', 'cherry'];
const keys = fruits.keys();
for (const key of keys) {
  console.log(key);
}
```

**Output:** 0, 1, 2

**5. values():** Returns an iterator containing the values of the array.

**Syntax:** array.values()

**Example:**

```
const fruits = ['apple', 'banana', 'cherry'];
const values = fruits.values();
for (const value of values) {
  console.log(value);
}
```

**Output:** apple, banana, cherry

**6. entries():** Returns an iterator containing key-value pairs for each element in the array.

**Syntax:** array.entries()

**Example:**

```
const fruits = ['apple', 'banana', 'cherry'];
const entries = fruits.entries();
for (const [key, value] of entries) {
  console.log(key, value);
}
```

**Output:** 0 apple, 1 banana, 2 cherry

**7. length:** A property (not a method) that provides the number of elements in an array.

**Syntax:** array.length

**Example:**

```
const fruits = ['apple', 'banana', 'cherry'];
console.log(fruits.length); // 3
```

# **STRING**

- string is a sequence of characters enclosed within single quotes ('') or double quotes (""). Strings are used to represent text and one of the primitive data types in JavaScript. **strings are immutable.** This means that once a string is created, its content cannot be changed.

### **Example-1:**

```
let str1 = 'Research, Innovation!& Discovery';
let str2 = "JavaScript is awesome!";
```

### **Example-2:**

```
let Name1 = "wit" // Double quotes
let Name2 = 'RID' // Single quotes
```

- You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

### **Example-3:**

```
let answer1 = "It's alright";
let answer2 = "He is called 'Hi'";
let answer3 = 'He is called "Hello"'
```

### **❖ Escape Character:**

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash
\b	Backspace	
\n	New Line	
\t	Horizontal Tabulator	

### **Example:**

1. **Single Quote (\'):** Used to include a single quote within a string delimited by single quotes.

```
let a= 'He said \'Hello\'';
console.log(a); Output: He said 'Hello'
```

2. **Double Quote (\"):** Used to include double quotes within a string delimited by double quotes.

```
let a = "She said \"Hi\"";
console.log(a); Output: She said "Hi"
```

3. **Backslash (\\\):** Used to include a backslash character in a string.

```
let a = 'This is a backslash: \\'';
console.log(a); Output: This is a backslash: \
```

4. **Backspace (\b):** Moves the cursor one character back.

```
let b = 'Hello\bWorld';
console.log(b); Output: HellWorld
```

5. **New Line (\n):** Inserts a new line.

```
let n = 'Hello\nWorld';
console.log(n); Output: Hello
World
```

6. **Horizontal Tabulator (\t):** Inserts a horizontal tab.

```
let h = 'Hello\tWorld';
console.log(h); Output: Hello    World
```

**Output:** Hello

World

### ❖ String Methods:

- 1) String length
- 2) String slice()
- 3) String substring()
- 4) String substr()
- 5) String replace()
- 6) String replaceAll()
- 7) String toUpperCase()
- 8) String toLowerCase()
- 9) String concat()
- 10) String trim()
- 11) String trimStart()
- 12) String trimEnd()
- 13) String padStart()
- 14) String padEnd()
- 15) String charAt()
- 16) String charCodeAt()
- 17) String split()

**1. String length (length property):** Returns the length of the string.

**Syntax:** string.length

**Example:** const str = "Research, Innovation";

console.log("length of string=", str.length); **Output:** length of string= 20

**2. String slice(start, end):** Extracts a portion of the string and returns it as a new string.

- Extracts a portion of the string from start (**inclusive**) to end (**exclusive**). By default end index will last
- Supports negative indices (counts from the end of the string if negative).
- Does not swap start and end if start > end.

**Syntax:** string.slice(startIndex, endIndex)

**Example:** let str = "RidBharat";

```
    console.log(str.slice(0, 4));
    console.log(str.slice(2, -3));
    console.log(str.slice(2, 3));
    console.log(str.slice(7, 4));
```

**Output:** RidB

dBha

d

<empty-string>

#### Ex-1

```
let str = "ridbharat";
console.log(str[-1]); // Output: undefined (negative indexing is not supported)
```

#### Ex-2

```
let str = "ridbhart";
console.log(str.slice(-1)); // Output: "t" (last character)
console.log(str.slice(-4, -1)); // Output: "har" (4th to last to 2nd to last character)
```

```
let sentence = "JavaScript is a versatile language.;"
```

**Extracting the word "versatile"**

```
let word = sentence.slice(16, 25);
```

```
console.log(word); Output: "versatile"
```

**Using negative indices to extract the word "language"**

```
let wordFromEnd = sentence.slice(-9, -1);
```

```
console.log(wordFromEnd); Output: "language"
```

**Extracting the substring "JavaScript" using only the start index**

```
let startOnly = sentence.slice(0);
```

```
console.log(startOnly); Output: "JavaScript is a versatile language."
```



### 3. String substring(startIndex, endIndex):

- Extracts a portion of the string from start (inclusive) to end (exclusive).
- Does not support negative indices;** negative values are treated as 0.
- Automatically swaps start and end if start > end.

**Syntax:** string.substring(startIndex, endIndex)

**Example:**

```
let str = "JavaScript";
console.log(str.substring(0, 4)); Output: "Java"
console.log(str.substring(4, 0)); Output: "Java" (start and end swapped)
console.log(str.substring(4, -3)); Output: "Java" (-3 treated as 0)
```

**Output:** Java

Java

Java

**Example:** let quote = "The quick brown fox jumps over the lazy dog";

**Extracting the word "quick"**

```
let word1 = quote.substring(4, 9);
console.log(word1); Output: "quick"
```

**Extracting from the start when `end` is omitted**

```
let fromStart = quote.substring(10);
console.log(fromStart); Output: "brown fox jumps over the lazy dog"
```

**Swapping `start` and `end` automatically**

```
let swapped = quote.substring(9, 4);
console.log(swapped); Output: "quick" (start and end are swapped)
```

**Using negative index (treated as 0)**

```
let negativeIndex = quote.substring(-5, 9);
console.log(negativeIndex); Output: "The quick" (-5 is treated as 0)
```

**Extracting the word "lazy"**

```
let word2 = quote.substring(35, 39);
console.log(word2); // Output: "lazy"
```

#### Key Differences:

Feature	slice(start, end)	substring(start, end)
Negative Indices	Supported	Treated as 0
Swaps start & end	No	Yes
Behavior When start > end	Returns empty string	Swaps the indices

#### Choosing Between Them:

- Use slice when you want negative indices or prefer stricter control over the arguments.
- Use substring if you're working with non-negative indices and want automatic index swapping for convenience.

#### **4. String substr(start, length):**

- Extracts a specified number of characters from a string, starting at the specified index.

**Syntax:** string.substr(startIndex, length)

**Example:**

```
const str = "Hello, World!";
const s = str.substr(7, 5);
console.log(s);
```

**Output:** "World"

#### **5. String replace oldValue, newValue):**

- Replaces the first occurrence of oldValue with newValue.

**Syntax:** string.replace(searchValue, replaceValue)

**Example:**

```
let a="RID BHARAT BHOPAL"
B=a.replace("BHOPAL", "Patna")
console.log(B)
```

**Output:** RID BHARAT Patna

#### **6. String replaceAll oldValue, newValue) (Available in ES2021 and later):**

- Replaces all occurrences of oldValue with newValue.

**Syntax:** string.replaceAll(searchValue, replaceValue)

**Example:**

```
const s="abababbababaababaab"
let ns=s.replaceAll("a","A")
console.log(ns)
```

**Output:** AbAbAbbAbAbAAbAbAAb

#### **7. String toUpperCase():**

- The toUpperCase() method converts all the characters in a string to uppercase. It does not modify the original string; instead, it returns a new string with all letters converted to uppercase.

**Syntax:** string.toUpperCase()

**Example:**

```
const s="rid bharat bhopal"
let ns=s.toUpperCase()
console.log(ns)
```

**Output:** RID BHARAT BHOPAL

#### **8. String toLowerCase():**

- toLowerCase() method converts all the characters in a string to lowercase. Like toUpperCase(), it does not modify the original string but returns a new string with all letters converted to lowercase.

**Syntax:** string.toLowerCase()

**Example:**

```
const s="RID BHARAT BHOPAL"
let ns=s.toLowerCase()
console.log(ns)
```

**Output:** rid bharat Bhopal

### Note-1: How to write first letter of a string in capital letter remaining should be lowerCase

#### Example:

```
function CFL(s) {  
    if (!s) {  
        return s;  
    }  
    else{  
        return s.charAt(0).toUpperCase() + s.slice(1).toLowerCase();  
    }  
}  
  
const user_str= "rid bharat Bhopal";  
const res = CFL(user_str);  
console.log(res); Output: Rid bharat bhopal
```

#### charAt()

- charAt() method is used to retrieve the character at a specified index in a string. The index is zero-based, meaning the first character is at index 0.

**Syntax:** string.charAt(index)

#### Example

```
const str = "JavaScript";  
const firstChar = str.charAt(0);  
// Gets the first character  
const fourthChar = str.charAt(3);  
// Gets the fourth character
```

### Note-2: How to write each first letter of a word in a string in capital letter remains should be lowerCase

#### Example:

```
function capitalizeWords(str) {  
    const words = str.split(' ').map(word => word[0].toUpperCase() + word.slice(1).toLowerCase());  
    return words.join(' ');\n}  
  
console.log(capitalizeWords("hello, WORLD! welcome to JAVASCRIPT."));
```

**Output:** "Hello, World! Welcome To Javascript."

#### Explanation of the Simplified Code:

1. str.split(' '): Splits the string into an array of words.
2. map(): For each word:
  - o word[0].toUpperCase(): Converts the first character to uppercase. The ?. ensures it works even if the word is empty.
  - o word.slice(1).toLowerCase(): Converts the rest of the word to lowercase.
3. join(' '): Combines the words back into a single string with spaces.

#### 2<sup>nd</sup> Method:

```
function capitalizeWords(str) {  
    const words = str.split(' ').map(word => {  
        const firstChar = word[0].toUpperCase();  
        const restChars = word.slice(1).toLowerCase();  
        return firstChar + restChars;  
    });  
    return words.join(' ');\n}  
  
console.log(capitalizeWords("hello, WORLD! welcome to JAVASCRIPT."));
```

**Note:** If you remove the ?, the function will still work correctly, provided there are no empty strings in the array or edge cases.

**9. String concat():** Combines one or more strings and returns a new string.

**Syntax:** string.concat(string1, string2, ..., stringN)

**Example:**

```
const str1 = "Hello, ";
const str2 = "World!";
const combinedStr = str1.concat(str2);
console.log(combinedStr); Output: "Hello, World!"
```

**10. String trim():** Removes whitespace from the beginning and end of the string.

**Syntax:** string.trim()

**Example:**

```
const str = " Hello, World! ";
const trimmedStr = str.trim();
console.log("String trim:", trimmedStr); Output: "Hello, World!"
```

**11. String trimStart() or String trimLeft():** Removes whitespace from beginning of string.

**Syntax:** string.trimStart()

**Example:**

```
const str = " Hello, World! ";
const trimmedStartStr = str.trimStart();
console.log("String trimStart:", trimmedStartStr); Output: "Hello, World! "
```

**12. String trimEnd() or String trimRight():** Removes whitespace from the end of the string.

**Syntax:** string.trimEnd()

**Example:**

```
const str = " Hello, World! ";
const trimmedEndStr = str.trimEnd();
console.log("String trimEnd:", trimmedEndStr); Output: " Hello, World!"
```

#### ❖ How to replace the space between the word as well as letter

##### 1. Using replaceAll() (ES2021+)

- **replaceAll()** method replaces all occurrences of a substring or pattern in the string.

**Example:**

```
let str = "H e l l o W o r l d";
let result = str.replaceAll(" ", ""); // Replace all spaces
console.log(result); Output: "HelloWorld"
```

##### 2. Remove the space between word

**Example:**

```
let str = "This is a string with multiple spaces.";
```

#### Split the string into an array of words

```
let wordsArray = str.split(' ');
```

#### Filter out empty strings

```
let filteredArray = wordsArray.filter(word => word !== "");
```

#### Join the array back into a string with a single space between words

```
let newStr = filteredArray.join(' ');
```

```
console.log(newStr); Output: "This is a string with multiple spaces."
```

### 13. String padStart(targetLength, padString):

- Pads the string with a specified character (or space) until it reaches the desired length.

**Syntax:** string.padStart(targetLength, padString)

**Example:** const str = "5";

```
const paddedStartStr = str.padStart(3, "0");
```

```
console.log("String padStart:", paddedStartStr); Output: "005"
```

### 14. String padEnd(targetLength, padString):

- Pads string from the end with a specified character (or space) until it reaches the desired length.

**Syntax:** string.padEnd(targetLength, padString)

**Example:** const str = "5";

```
const paddedEndStr = str.padEnd(3, "0");
```

```
console.log("String padEnd:", paddedEndStr); Output: "500"
```

### 15. String charAt(index):

Returns the character at the specified index. charAt is useful for accessing individual characters

**Syntax:** string.charAt(index)

**Example:** const str = "Hello, World!";

```
const char = str.charAt(7);
```

```
console.log("String charAt:", char); Output: "W"
```

### 16. String charCodeAt(index):

- Returns the Unicode value (integer) of the character at the specified index.

**Syntax:** string.charCodeAt(index)

**Example:** const str = "Hello, World!";

```
const charCode = str.charCodeAt(7);
```

```
console.log("String charCodeAt:", charCode); Output: 87
```

### 17. String split(separator):

- Splits the string into an array of substrings based on the specified separator.

**Syntax:** string.split(separator, limit)

**Example:** const str = "apple,banana,cherry";

```
const fruits = str.split(",");
```

```
console.log("String split:", fruits); Output: ["apple", "banana", "cherry"]
```

#### Example 1: Splitting a Sentence into Words

```
const sentence = "The quick brown fox jumps over the lazy dog";
```

```
const words = sentence.split(" ");
```

```
console.log("String split:", words);
```

```
Output: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]
```

#### Example 2: Splitting a URL into Components

```
const url = "https://www.example.com/path/to/page";
```

```
const components = url.split("/");
```

```
console.log("String split:", components);
```

```
Output: ["https:", "", "www.example.com", "path", "to", "page"]
```

#### Additional Example: Splitting with Limit

```
const str = "one,two,three,four,five";
```

```
const limitedSplit = str.split(",", 3);
```

```
console.log("String split with limit:", limitedSplit);
```

```
Output: ["one", "two", "three"]
```



**Example:**

- String split(separator)  
conststrSplit = "apple,banana,cherry".split(",");  
console.log("String split:", strSplit); // Output: ["apple", "banana",  
let s="hello to all"  
console.log(s[11])//prints the element stored at index 11  
console.log(s.length)//prints the length of string  
console.log(s.charAt(1))//prints the element stored at index 1  
let res=s.toUpperCase()  
console.log(res)  
console.log(s.includes("to"))//returns true if it is present in string else false will be returned  
console.log(s.startsWith("h"))//returns true if starting element is present  
console.log(s.endsWith("all"))  
console.log(s.replace("", "\$"))//it will add \$ at the start  
console.log(s.replace("hello", "\$"))  
console.log(s.replace("h", "\$"))//replaces only the first occurrence  
console.log(s.replaceAll("l", "-"))//replaces all the occurrences  
console.log(s[9])  
console.log(s.charCodeAt(9))//display UNICODE value for given element  
console.log(String.fromCharCode(65))  
let r="\*".repeat(3)//repeats the given string value 3 times  
console.log(r)  
console.log(s.slice(0,3))//equivalent to slicing operation  
console.log(s.slice(-3))  
//Negative values are not supported in substring method  
console.log(s.substring(0,5))//the first value indicates initial index position & second value indicates last index position  
console.log(s.substring(5))  
console.log(s.substr(9,3))//the first value indicates index position & second value indicates length of substring to be displayed  
console.log(s.substr(9,3))  
console.log(s.split())//o/p [ 'hello to all' ]  
console.log(s.split(" "))// o/p [ 'hello', 'to', 'all' ]  
console.log(s.split(""))

**Output:**

```
[  
  'h', 'e', 'l', 'l',  
  'o', ' ', 't', 'o',  
  ' ', 'a', 'l', 'l'  
]  
let r=s.split(" ")  
console.log(r.join(""))//o/p hellotoall  
console.log(r.join(" "))//o/p hello to all
```



**1. Program to print the count of each element from given string**

```
const s="ridbhartbhopal"  
let d={}  
for(let i of s){  
    if(d[i]==undefined)  
        d[i]=1  
    else  
        d[i]+=1  
}  
console.log(d) output: { r: 2, i: 1, d: 1, b: 2, h: 2, a: 2, t: 1, o: 1, p: 1, l: 1 }
```

**2. Program to print the count of given element from given string**

```
const prompt = require('prompt-sync')();  
const s = "ridbhartbhopal"; // The string to search in  
let ele = prompt("Enter the element to be searched- "); // Prompting user input  
let ct = 0;  
for (let i of s) {  
    if (i === ele) {  
        ct += 1;  
    }  
}  
console.log(`The count of '${ele}' in the given string is ${ct}`);  
Output: Enter the element to be searched- a  
The count of 'a' in the given string is 2
```

**3. Program to print the reverse of string**

```
let rev=""  
for(let i of s){  
    rev=i+rev  
}  
console.log(rev)
```

**4. Program to check whether given string is palindrome or not**

```
const s = "madam";  
let rev = "";  
for (let i of s) {  
    rev = i + rev;  
}  
if (rev === s) {  
    console.log("Palindrome");  
} else {  
    console.log("Not palindrome");  
}
```

**Output:** Palindrome

### ❖ String Search Methods:

- 1) String indexOf()
- 2) String lastIndexOf()
- 3) String search()
- 4) String match()
- 5) String matchAll()
- 6) String includes()
- 7) String startsWith()
- 8) String endsWith()

#### 1. String indexOf(substring, startIndex):

- Returns the index of the first occurrence of a substring, or -1 if not found..

**Syntax:** let index = str.indexOf(searchValue, fromIndex);

**Example-1:** const str = "Hello, World!";

```
const index = str.indexOf("World");
console.log("String indexOf:", index); Output: 7
```

#### Example 2: Finding a Word in a Sentence

```
const sentence = "The quick brown fox jumps over the lazy dog.";
const index = sentence.indexOf("fox");
console.log("String indexOf:", index); Output: 16
```

#### Example 3: Finding a Substring in a Long String with Start Index

```
const text = "She sells sea shells by the sea shore.";
const index = text.indexOf("sea", 10);
console.log("String indexOf:", index); Output: 27
```

#### Example 4: Finding a Character in a String of Numbers

```
const numbers = "12345678901234567890";
const index = numbers.indexOf("5");
console.log("String indexOf:", index); Output: 4
```

#### 2. String lastIndexOf(substring, startIndex):

- Returns the index of the last occurrence of a substring, or -1 if not found.

**Syntax:** let lastIndex = str.lastIndexOf(searchValue, fromIndex);

**Example:** const str = "Hello, World, World!";

```
const lastIndex = str.lastIndexOf("World");
console.log("String lastIndexOf:", lastIndex) Output: 13
```

#### Example 1: Finding the Last Occurrence of a Character

```
const sentence = "abracadabra";
const lastIndex = sentence.lastIndexOf("a");
console.log("String lastIndexOf:", lastIndex); Output: 10
```

#### Example 2: Finding the Last Occurrence of a Word in a Sentence

```
const sentence = "The quick brown fox jumps over the lazy dog. The quick brown fox is fast.";
const lastIndex = sentence.lastIndexOf("quick");
console.log("String lastIndexOf:", lastIndex); Output: 40
```

#### Example 3: Finding the Last Occurrence of a Substring with Start Index.

```
const text = "This is a test. This is only a test. This is a final test.";
const lastIndex = text.lastIndexOf("test", 30);
console.log("String lastIndexOf:", lastIndex); Output: 24
```



### 3. String search(regexp):

- Searches the string for a specified pattern (regular expression) and returns the index of the first match, or -1 if not found.

**Syntax:** let position = str.search(regexp);

**Example:** const str = "The cat and the hat";

const index = str.search(/cat/);

console.log("String search:", index); **Output:** 4

### 4. String match(regexp):

- Searches string for a specified pattern (regular expression) and returns an array of the matches.

**Syntax:** let matches = str.match(regexp);

**Example:** const str = "The cat and the hat";

const matches = str.match(/(cat|hat)/g);

console.log("String match:", matches); **Output:** ["cat", "hat"]

### 5. String matchAll(regexp):

- Searches the string for a specified pattern (regular expression) and returns an iterable containing all matches and their capture groups.

**Syntax:** let iterator = str.matchAll(regexp);

**Example:** const str = "The cat and the hat";

const matchIterator = str.matchAll(/(cat|hat)/g);

for (const match of matchIterator) {

console.log("String matchAll:", match[0]);

}

**Output:** "cat"

"hat"

### 6. String includes(substring):

- Checks if the string contains the specified substring and returns a boolean.

**Syntax:** let result = str.includes(searchValue, fromIndex);

**Example-1:** const str = "Hello, World!";

const includes = str.includes("World");

console.log("String includes:", includes); **Output:** true

**Example-2:**

const message = "I like apples.;"

const hasApples = message.includes("apples");

console.log("Has apples:", hasApples); **Output:** true

**Example-3:**

const text = "Learning is fun!";

const hasFun = text.includes("FUN"); // Note the uppercase "FUN"

console.log("Has fun:", hasFun); **Output:** false

**Example-4:**

const line = "The cat is sleeping. The cat is happy.;"

const hasCat = line.includes("cat", 20); // Start searching from index 20

console.log("Has cat after index 20:", hasCat); **Output:** true

## 7. String startsWith(substring):

- Checks if the string starts with the specified substring and returns a boolean.

**Syntax:** let result = str.startsWith(searchString, position);

### Example-1:

```
const str = "Hello, World!";
conststartsWith = str.startsWith("Hello");
console.log("String startsWith:", startsWith); Output: true
```

### Example-2:

```
const sentence = "Good morning, everyone!";
const startsWithGood = sentence.startsWith("Good");
console.log("Starts with 'Good':", startsWithGood); Output: true
```

### Example-3:

```
const phrase = "JavaScript is amazing!";
const startsWithJava = phrase.startsWith("javascript"); // Note the lowercase "j"
console.log("Starts with 'javascript':", startsWithJava); Output: false
```

### Example-4:

```
const text = "Learning to code is fun.";
const startsWithCode = text.startsWith("code", 12); // Start checking from index 12
console.log("Starts with 'code' from index 12:", startsWithCode); Output: true
```

## 8. String endsWith(substring):

- Checks if the string ends with the specified substring and returns a boolean.

- Description:** Checks if a string ends with a specified value.

**Syntax:** let result = str.endsWith(searchString, length);

### Example-1:

```
const str = "Hello, World!";
constendsWith = str.endsWith("World!");
console.log("String endsWith:", endsWith); Output: true
```

### Example-2:

```
const sentence = "I love programming.";
const endsWithProgramming = sentence.endsWith("programming.");
console.log("Ends with 'programming':", endsWithProgramming); Output: true
```

### Example-3:

```
const phrase = "JavaScript is fun!";
const endsWithFun = phrase.endsWith("FUN!"); // Note the uppercase "FUN"
console.log("Ends with 'FUN!':", endsWithFun); Output: false
```

### Example-4:

```
const text = "Learning is enjoyable.";
const endsWithLearning = text.endsWith("Learning", 8); // Only consider the first 8 characters
console.log("Ends with 'Learning' in first 8 characters:", endsWithLearning); Output: true
```

# **THIS KEYWORD**

- This keyword is a special variable that is automatically created in the scope of every function.
- The this keyword is a reference variable that refers to the current object.
- Which object depends on how this is being invoked (used or called).
- The this keyword refers to different objects depending on how it is used:
- In an object method, this refers to the object.
- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In a function, in strict mode, this is undefined.
- In an event, this refers to the element that received the event.
- Methods like call(), apply(), and bind() can refer this to any object.

## **1. Global Context:**

- In the global context (outside of any function or object), this refers to the global object, which is window in a browser environment and global in Node.js.

### **Example:**

```
console.log(this === window); // true (in a browser)
console.log(this === global); // true (in Node.js)
```

## **2. Function Context:**

- In a regular function, this refers to the object that called the function. It can change depending on how the function is invoked.

### **Example:**

```
function greet() {
  console.log(`Hello, ${this.name}`);
}
const person = { name: "Sangam" };
person.greet = greet;
person.greet(); // Outputs "Hello, Sangam!"
const anotherGreet = person.greet;
anotherGreet(); // Outputs "Hello, undefined!"
```

**Note:** In the first call to person.greet(), this inside the greet function refers to the person object. In the second call, when anotherGreet is called without any context, this refers to the global object (window in a browser), so this.name is undefined.

## **3. Arrow Functions:**

- Arrow functions have a fixed this value. They inherit this from the surrounding lexical context, which means they use this value of the enclosing function or scope.

### **Example:**

```
const person = {
  name: "Alice",
  greet: function () {
    setTimeout(() => {
      console.log(`Hello, ${this.name}`);
    }, 1000);
  }
};
person.greet(); // Outputs "Hello, Alice!" after a 1-second delay
```

**Note:** In this example, the arrow function inside the setTimeout callback captures this from the person object because it's defined within the greet method.



#### **4. Constructor Functions:**

- When a function is used as a constructor with the new keyword, this refers to the newly created object.

##### **Example:**

```
function Person(name) {  
    this.name = name;  
}  
  
const person = new Person("raj");  
console.log(person.name); // Outputs "raj"
```

**Note:** In this case, this inside the Person constructor refers to the newly created person object.

#### **5. Event Handlers:**

- In the context of event handlers (e.g., for DOM elements), this refers to the DOM element that triggered the event.

##### **Example (HTML and JavaScript):**

```
<button id="myButton">Click me</button>  
<script>  
const button = document.getElementById("myButton");  
button.addEventListener("click", function () {  
    console.log(this.textContent); // Outputs "Click me"  
});  
</script>
```

##### **Note:**

- In this example, this inside the event listener function refers to the button element because it triggered the click event.

#### **6. Global Object in Strict Mode:**

- In strict mode ("use strict"), when a function is called without any context (not as a method or constructor), this is undefined rather than referring to the global object.

##### **Example:**

```
"use strict";  
function strictFunction() {  
    console.log(this);  
}  
strictFunction(); // Outputs "undefined"
```

**Note:** this in JavaScript is a dynamic keyword whose value depends on how a function is invoked.

This Precedence

➤ To determine which object this refers to; use the following precedence of order.

##### **Precedence    Object**

- 1 bind()
- 2 apply() and call()
- 3 Object method
- 4 Global scope

- Is this in a function being called using bind()?
- Is this in a function being called using apply()?
- Is this in a function being called using call()?
- Is this in an object function (method)?
- Is this in a function in the global scope.

# HOISTING

- JavaScript hoisting is a behavior where variable and function declarations are moved to the top of their containing scope during the compilation phase before the code is executed.
- It is a mechanism in JavaScript that moves declaration of variables and functions at the top.

## ❖ Variable Hoisting:

### Example-1:

```
console.log(myVar); // Outputs: undefined  
var myVar = 10;  
console.log(myVar); // Outputs: 10
```

### Example-2:

```
console.log(a) //ReferenceError: Cannot access 'a' before initialization  
let a=6  
console.log(a) //output: 6
```

### Example-3:

```
console.log(a) //ReferenceError: Cannot access 'a' before initialization  
const a=6  
console.log(a) // 6
```

## ❖ Function Hoisting:

- Function declarations, unlike variable declarations, are fully hoisted. This means that both the function name and its implementation are moved to the top of the scope, making the function available for use before it's declared in the code.

### Example:

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

**Note:** heresayHello function are declred with function keyword so output come

Example:

```
myFunc(); // TypeError: myFunc is not a function  
var myFunc = function () {  
    console.log("Hello!");  
};
```

**Note:** themyFunc variable is hoisted, but it's not initialized as a function until the assignment statement is encountered. Thus, attempting to call it before the assignment results in a TypeError.

# DATE OBJECT

- Date object is used to work with dates and times. It allows you to represent and manipulate dates, perform date arithmetic, and format dates for display. The Date object is a core part of JavaScript and is commonly used in various web applications.

## ❖ Date Object Methods:

- The Date object provides various methods for working with dates and times:
  - 1) `getDate()`, `getMonth()`, `getFullYear()`: Get day, month (0-11), and year components of a date.
  - 2) `getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()`: Get the time components of a date.
  - 3) `getDay()`: Get the day of the week (0 = Sunday, 1 = Monday, ...).
  - 4) `toString()`, `toDateString()`, `toTimeString()`: Convert a date to a string.
  - 5) `toLocaleString()`, `toLocaleDateString()`, `toLocaleTimeString()`: Convert a date to a localized string.
  - 6) `getTime()`: Get the timestamp (milliseconds since January 1, 1970).
  - 7)  `setDate()`, `setMonth()`, `setFullYear()`: Set the day, month, and year components of a date.
  - 8) `setHours()`, `setMinutes()`, `setSeconds()`, `setMilliseconds()`: Set time components of a date.
  - 9)  `setTime()`: Set the timestamp.

## ❖ `getDate()`, `getMonth()`, `getFullYear()`:

- A. **`getDate()`:** This method returns the day of the month (from 1 to 31) for a specified date.
- B. **`getMonth()`:** This method returns the month (from 0 to 11) for a specified date. Note that January is 0, February is 1, and so on.
- C. **`getFullYear()`:** This method returns the year (a four-digit number) for a specified date.

### Example:

- **Note:** JavaScript's Date constructor does not natively support the dd-mm-yyyy format for date strings. It expects the date string in the yyyy-mm-dd format or mm/dd/yyyy format.

```
// let date=new Date("30-09-2023") // wrong
// let date=new Date("09-30-2023") // right
let date=new Date("2023-09-30")
let day=date.getDate()
let month1=date.getMonth()
let month=date.getMonth()+1
let year=date.getFullYear()
console.log("Date=", day)
console.log("Month=", month1)
console.log("Month=", month)
console.log("Year=", year)
```

**Output:**  
Date= 30  
Month= 8  
Month= 9  
Year= 2023

## ❖ `getHours()`, `getMinutes()`, `getSeconds()`, `getMilliseconds()`:

- A. **`getHours()`:** This method returns the hour (from 0 to 23) of a specified date and time.
- B. **`getMinutes()`:** This method returns the minutes (from 0 to 59) of a specified date and time.
- C. **`getSeconds()`:** This method returns the seconds (from 0 to 59) of a specified date and time.
- D. **`getMilliseconds()`:** This method returns the milliseconds (from 0 to 999) of a specified date and time.

### Example:

```
let now = new Date();
```



```
let hours = now.getHours();
let minutes = now.getMinutes();
let seconds = now.getSeconds();
let milliseconds = now.getMilliseconds();
```

❖ **Determine AM or PM**

```
let ampm = hours >= 12 ? 'PM' : 'AM';
```

❖ **Convert hours from 24-hour format to 12-hour format**

```
hours = hours % 12;
```

```
hours = hours ? hours : 12; // the hour '0' should be '12'
```

❖ **Format minutes and seconds to be always two digits**

```
minutes = minutes < 10 ? '0' + minutes : minutes;
seconds = seconds < 10 ? '0' + seconds : seconds;
console.log("Hours:", hours);
console.log("Minutes:", minutes);
console.log("Seconds:", seconds);
console.log("Milliseconds:", milliseconds);
console.log(`${hours}:${minutes}:${seconds} ${ampm}`);
```

**Output:**

Hours: 12

Minutes: 35

Seconds: 17

Milliseconds

**Example-1 How to get the local date and time using the getHours(), getMinutes(), getSeconds(), and getMilliseconds():**

❖ Create a Date object for the current local date and time

```
const localDate = new Date();
console.log(localDate)
```

❖ Get the local time components

```
const hours = localDate.getHours();
const minutes = localDate.getMinutes();
const seconds = localDate.getSeconds();
const milliseconds = localDate.getMilliseconds(); // Output the local time components
console.log(`Local Hours: ${hours}`);
console.log(`Local Minutes: ${minutes}`);
console.log(`Local Seconds: ${seconds}`);
console.log(`Local Milliseconds: ${milliseconds}`);
```

**Output:**

2023-09-20T07:32:29.621Z

Local Hours: 13

Local Minutes: 2

Local Seconds: 29

Local Milliseconds: 621

1. **getDay()**: Get the day of the week (0 = Sunday, 1 = Monday, ...).
2. **toString(), toDateString(), toTimeString()**: Convert a date to a string.
3. **toLocaleString(), toLocaleDateString(), toLocaleTimeString()**: Convert a date to a localized string.
4. **getTime()**: Get the timestamp (milliseconds since January 1, 1970).
5.  **setDate(), setMonth(), setFullYear()**: Set the day, month, and year components of a date.



6. `setHours()`, `setMinutes()`, `setSeconds()`, `setMilliseconds()`: Set the time components of a date.
7. `getTime()`: Set the timestamp.

**Example-2:**

```
const date = new Date("2023-09-30T14:30:45.123Z");
const dayOfWeek = date.getDay();
const dateString = date.toString();
const dateOnlyString = date.toDateString();
const timeOnlyString = date.toTimeString();
const localizedString = date.toLocaleString();
const localizedDateString = date.toLocaleDateString();
const localizedTimeString = date.toLocaleTimeString();
const timestamp = date.getTime();
date.setDate(10);
date.setMonth(2);
date.setFullYear(2022);

// Set the time components of the date
date.setHours(18);
date.setMinutes(45);
date.setSeconds(30);
date.setMilliseconds(500);
date.setTime(1678554330500);
console.log(`Day of the Week: ${dayOfWeek}`);
console.log(`Date as String: ${dateString}`);
console.log(`Date Only String: ${dateOnlyString}`);
console.log(`Time Only String: ${timeOnlyString}`);
console.log(`Localized Date String: ${localizedString}`);
console.log(`Localized Date Only String: ${localizedDateString}`);
console.log(`Localized Time Only String: ${localizedTimeString}`);
console.log(`Timestamp: ${timestamp}`);
console.log(`Modified Date: ${date.toString()}`);
```

**Output:**

```
Day of the Week: 6
Date as String: Sat Sep 30 2023 20:00:45 GMT+0530 (India Standard Time)
Date Only String: Sat Sep 30 2023
Time Only String: 20:00:45 GMT+0530 (India Standard Time)
Localized Date String: 9/30/2023, 8:00:45 PM
Localized Date Only String: 9/30/2023
Localized Time Only String: 8:00:45 PM
Timestamp: 1696084245123
Modified Date: Sat Mar 11 2023 22:35:30 GMT+0530 (India Standard Time)
```

❖ **Set the timer in JavaScript:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Countdown Timer</title>
</head>
<body>
```



```
<div id="timer"></div>
<script>
```

#### **Set the initial time to 1 hour in milliseconds**

```
let totalTime = 60 * 60 * 1000; // for 1hr
let totalTime = 2*(60 * 60 * 1000); for 2 hr
```

#### **Update the timer every second**

```
let interval = setInterval(() => {
```

#### **Calculate hours, minutes, and seconds**

```
let hours = Math.floor(totalTime / (1000 * 60 * 60));
let minutes = Math.floor((totalTime % (1000 * 60 * 60)) / (1000 * 60));
let seconds = Math.floor((totalTime % (1000 * 60)) / 1000);
```

#### **Format minutes and seconds to be always two digits**

```
minutes = minutes < 10 ? '0' + minutes : minutes;
seconds = seconds < 10 ? '0' + seconds : seconds;
```

#### **Display the result in the timer div**

```
document.getElementById("timer").innerHTML = `${hours}:${minutes}:${seconds}`;
```

#### **If the timer reaches 0, stop the interval**

```
if (totalTime <= 0) {
    clearInterval(interval);
    document.getElementById("timer").innerHTML = "Time's up!";
}
```

#### **Decrease total time by 1 second**

```
totalTime -= 1000;
}, 1000);
</script>
</body>
</html>
```

#### **Output:**

2:59:03

# MATH IN JAVASCRIPT

- math object provides several constants and methods to perform mathematical operation.
- 1) **Math.abs(x):** Returns the absolute value of a number x.
- 2) **Math.acos(x):** Returns the arccosine (in radians) of a number x, where x is in the range [-1, 1].
- 3) **Math.asin(x):** Returns the arcsine (in radians) of a number x, where x is in the range [-1, 1].
- 4) **Math.atan(x):** Returns the arctangent (in radians) of a number x.
- 5) **Math.cbrt(x):** Returns the cube root of a number x.
- 6) **Math.ceil(x):** Returns the smallest integer greater than or equal to a number x.
- 7) **Math.cos(x):** Returns the cosine of a number x (assumed to be in radians).
- 8) **Math.cosh(x):** Returns the hyperbolic cosine of a number x.
- 9) **Math.exp(x):** Returns the exponential value of a number x.
- 10) **Math.floor(x):** Returns the largest integer less than or equal to a number x.
- 11) **Math.hypot(...args):** Returns the square root of the sum of squares of its arguments, effectively calculating the hypotenuse of a right triangle.
- 12) **Math.log(x):** Returns the natural logarithm (base e) of a number x.
- 13) **Math.max(...args):** Returns the maximum value among a list of numbers passed as arguments.
- 14) **Math.min(...args):** Returns the minimum value among a list of numbers passed as arguments.
- 15) **Math.pow(x, y):** Returns x raised to the power of y.
- 16) **Math.random():** Returns a random number between 0 (inclusive) and 1 (exclusive).
- 17) **Math.round(x):** Returns the value of a number x rounded to the nearest integer.
- 18) **Math.sign(x):** Returns sign of a number x (1 for +ve, -1 for negative, 0 for zero, and NaN for NaN).
- 19) **Math.sin(x):** Returns the sine of a number x (assumed to be in radians).
- 20) **Math.sinh(x):** Returns the hyperbolic sine of a number x.
- 21) **Math.sqrt(x):** Returns the square root of a number x.
- 22) **Math.tan(x):** Returns the tangent of a number x (assumed to be in radians).
- 23) **Math.tanh(x):** Returns the hyperbolic tangent of a number x.
- 24) **Math.trunc(x):** Returns the integer part of a number x (removing the decimal part).

**Example:**

## 1. Math.abs(x)

- **Syntax:** `Math.abs(x)`
- **Description:** Returns the absolute value of a number x.
- **Example:**

```
console.log(Math.abs(-10)); // Output: 10  
console.log(Math.abs(3.14)); // Output: 3.14
```

## 2. Math.acos(x)

- **Syntax:** `Math.acos(x)`
- **Description:** Returns the arccosine (in radians) of a number x, where x is in the range [-1, 1].
- **Example:**

```
console.log(Math.acos(1)); // Output: 0  
console.log(Math.acos(0)); // Output: 1.5707963267948966 ( $\pi/2$ )
```

## 3. Math.asin(x)

- **Syntax:** `Math.asin(x)`
- **Description:** Returns the arcsine (in radians) of a number x, where x is in the range [-1, 1].

- **Example:**

```
console.log(Math.asin(1)); // Output: 1.5707963267948966 ( $\pi/2$ )
console.log(Math.asin(0)); // Output: 0
```

#### 4. Math.atan(x)

- **Syntax:** Math.atan(x)

- **Description:** Returns the arctangent (in radians) of a number x.

- **Example:**

```
console.log(Math.atan(1)); // Output: 0.7853981633974483 ( $\pi/4$ )
console.log(Math.atan(0)); // Output: 0
```

#### 5. Math.cbrt(x)

- **Syntax:** Math.cbrt(x)

- **Description:** Returns the cube root of a number x.

- **Example:**

```
console.log(Math.cbrt(27)); // Output: 3
console.log(Math.cbrt(-8)); // Output: -2
```

#### 5. Math.ceil(x)

**Syntax:** Math.ceil(x)

**Description:** Returns the smallest integer greater than or equal to a number x.

**Example:**

```
console.log(Math.ceil(4.2)); // Output: 5
console.log(Math.ceil(-1.7)); // Output: -1
```

#### 7. Math.cos(x)

- **Syntax:** Math.cos(x)

- **Description:** Returns the cosine of a number x (assumed to be in radians).

- **Example:**

```
console.log(Math.cos(0)); // Output: 1
console.log(Math.cos(Math.PI)); // Output: -1
```

#### 8. Math.cosh(x)

- **Syntax:** Math.cosh(x)

- **Description:** Returns the hyperbolic cosine of a number x.

- **Example:**

```
console.log(Math.cosh(0)); // Output: 1
console.log(Math.cosh(1)); // Output: 1.5430806348152437
```

#### 9. Math.exp(x)

- **Syntax:** Math.exp(x)

- **Description:** Returns the exponential value of a number x, i.e.,  $e^x$ .

- **Example:**

```
console.log(Math.exp(1)); // Output: 2.718281828459045 (e)
console.log(Math.exp(0)); // Output: 1
```

#### 10. Math.floor(x)

- **Syntax:** Math.floor(x)

- **Description:** Returns the largest integer less than or equal to a number x.

- **Example:**

```
console.log(Math.floor(4.7)); // Output: 4
console.log(Math.floor(-1.2)); // Output: -2
```



## 11. Math.hypot(...args)

- **Syntax:** Math.hypot(...args)
- **Description:** Returns the square root of the sum of squares of its arguments, effectively calculating the hypotenuse of a right triangle.
- **Example:**

```
console.log(Math.hypot(3, 4)); // Output: 5 (sqrt(3^2 + 4^2))  
console.log(Math.hypot(1, 2, 3)); // Output: 3.7416573867739413 (sqrt(1^2 + 2^2 + 3^2))
```

## 12. Math.log(x)

- **Syntax:** Math.log(x)
- **Description:** Returns the natural logarithm (base e) of a number x.
- **Example:**

```
console.log(Math.log(1)); // Output: 0 (ln(1))  
console.log(Math.log(Math.E)); // Output: 1 (ln(e))
```

## 13. Math.max(...args)

- **Syntax:** Math.max(...args)
- **Description:** Returns the maximum value among a list of numbers passed as arguments.
- **Example:**

```
console.log(Math.max(1, 2, 3, 4, 5)); // Output: 5  
console.log(Math.max(-1, -2, -3)); // Output: -1
```

## 14. Math.min(...args)

- **Syntax:** Math.min(...args)
- **Description:** Returns the minimum value among a list of numbers passed as arguments.
- **Example:**

```
console.log(Math.min(1, 2, 3, 4, 5)); // Output: 1  
console.log(Math.min(-1, -2, -3)); // Output: -3
```

## 15. Math.pow(x, y)

- **Syntax:** Math.pow(x, y)
- **Description:** Returns x raised to the power of y.
- **Example:**

```
console.log(Math.pow(2, 3)); // Output: 8 (2^3)  
console.log(Math.pow(5, 0)); // Output: 1 (5^0)
```

## 16. Math.random()

- **Syntax:** Math.random()
- **Description:** Returns a random number between 0 (inclusive) and 1 (exclusive).
- **Example:**

```
console.log(Math.random()); // Output: A random number between 0 and 1
```

## 17. Math.round(x)

- **Syntax:** Math.round(x)
- **Description:** Returns the value of a number x rounded to the nearest integer.
- **Example:**

```
console.log(Math.round(4.7)); // Output: 5  
console.log(Math.round(4.4)); // Output: 4
```

## 18. Math.sign(x)

- **Syntax:** Math.sign(x)

- **Description:** Returns the sign of a number x (1 for positive, -1 for negative, 0 for zero, and NaN for NaN).
- **Example:**

```
console.log(Math.sign(10)); // Output: 1  
console.log(Math.sign(-10)); // Output: -1  
console.log(Math.sign(0)); // Output: 0  
console.log(Math.sign(NaN)); // Output: NaN
```

## 19. Math.sin(x)

- **Syntax:** Math.sin(x)
- **Description:** Returns the sine of a number x (assumed to be in radians).
- **Example:**

```
console.log(Math.sin(0)); // Output: 0  
console.log(Math.sin(Math.PI / 2)); // Output: 1
```

## 20. Math.sinh(x)

- **Syntax:** Math.sinh(x)
  - **Description:** Returns the hyperbolic sine of a number x.
  - **Example:**
- ```
console.log(Math.sinh(0)); // Output: 0  
console.log(Math.sinh(1)); // Output: 1.1752011936438014
```

## 21. Math.sqrt(x)

- **Syntax:** Math.sqrt(x)
  - **Description:** Returns the square root of a number x.
  - **Example:**
- ```
console.log(Math.sqrt(9)); // Output: 3  
console.log(Math.sqrt(16)); // Output: 4
```

## 22. Math.tan(x)

- **Syntax:** Math.tan(x)
  - **Description:** Returns the tangent of a number x (assumed to be in radians).
  - **Example:**
- ```
console.log(Math.tan(0)); // Output: 0  
console.log(Math.tan(Math.PI / 4)); // Output: 1
```

## 23. Math.tanh(x)

- **Syntax:** Math.tanh(x)
  - **Description:** Returns the hyperbolic tangent of a number x.
  - **Example:**
- ```
console.log(Math.tanh(0)); // Output: 0  
console.log(Math.tanh(1)); // Output: 0.7615941559557649
```

## 24. Math.trunc(x)

- **Syntax:** Math.trunc(x)
  - **Description:** Returns the integer part of a number x (removing the decimal part).
  - **Example:**
- ```
console.log(Math.trunc(4.9)); // Output: 4  
console.log(Math.trunc(-4.9)); // Output: -4
```

# **NUMBER OBJECT**

- The JavaScript number object enables you to represent a numeric value.
- 1) **Number.isNaN(value):** Checks if a value is NaN.
- 2) **Number.isFinite(value):** Checks if a value is finite (not NaN, Infinity, or -Infinity).
- 3) **Number.parseInt(string, radix):** Parses a string and returns an integer.
- 4) **Number.parseFloat(string):** Parses a string and returns a floating-point number.
- 5) **Number.toFixed(digits):** Converts a number to a string with a fixed number of decimal places.
- 6) **Number.toPrecision(precision):** Converts a number to a string with a specified number of significant digits.
- 7) **Number.toString(base):** Converts a number to a string in a specified base.
- 8) **Number.isInteger(value):** Checks if a value is an integer.
- 9) **Number.MAX\_VALUE:** Represents the maximum numeric value.
- 10) **Number.MIN\_VALUE:** Represents the smallest positive numeric value.
- 11) **Number.POSITIVE\_INFINITY:** Represents positive infinity.
- 12) **Number.NEGATIVE\_INFINITY:** Represents negative infinity.

## **1. Number.isNaN(value)**

- **Syntax:** Number.isNaN(value)
- **Description:** Checks if a value is NaN (Not-a-Number).
- **Example:**

```
console.log(Number.isNaN(NaN)); // Output: true
console.log(Number.isNaN(123)); // Output: false
```

## **2. Number.isFinite(value)**

- **Syntax:** Number.isFinite(value)
- **Description:** Checks if a value is finite (not NaN, Infinity, or -Infinity).
- **Example:**

```
console.log(Number.isFinite(123)); // Output: true
console.log(Number.isFinite(Infinity)); // Output: false
```

## **3. Number.parseInt(string, radix)**

- **Syntax:** Number.parseInt(string, radix)
- **Description:** Parses a string and returns an integer. The radix is an integer between 2 and 36 that represents the base of the string.
- **Example:**

```
console.log(Number.parseInt('100', 10)); // Output: 100
console.log(Number.parseInt('11', 2)); // Output: 3
```

## **4. Number.parseFloat(string)**

- **Syntax:** Number.parseFloat(string)
- **Description:** Parses a string and returns a floating-point number.
- **Example:**

```
console.log(Number.parseFloat('3.14')); // Output: 3.14
console.log(Number.parseFloat('123abc')) // Output: 123
```

## **5. Number.toFixed(digits)**

- **Syntax:** number.toFixed(digits)

- **Description:** Converts a number to a string with a fixed number of decimal places.
- **Example:** let num = 3.14159;  
console.log(num.toFixed(2)); // Output: '3.14'  
console.log(num.toFixed(4)); // Output: '3.1416'

## 6. Number.toPrecision(precision)

- **Syntax:** number.toPrecision(precision)
- **Description:** Converts a number to a string with a specified number of significant digits.
- **Example:** let num = 3.14159;  
console.log(num.toPrecision(2)); // Output: '3.1'  
console.log(num.toPrecision(4)); // Output: '3.142'

## 7. Number.toString(base)

- **Syntax:** number.toString(base)
- **Description:** Converts a number to a string in a specified base. The base is an integer between 2 and 36.
- **Example:** let num = 255;  
console.log(num.toString(16)); // Output: 'ff'  
console.log(num.toString(2)); // Output: '11111111'

## 8. Number.isInteger(value)

- **Syntax:** Number.isInteger(value)
- **Description:** Checks if a value is an integer.
- **Example:** console.log(Number.isInteger(123)); // Output: true  
console.log(Number.isInteger(123.45)); // Output: false

## 9. Number.MAX\_VALUE

- **Syntax:** Number.MAX\_VALUE
- **Description:** Represents the maximum numeric value that can be represented in JavaScript.
- **Example:** console.log(Number.MAX\_VALUE); // Output: 1.7976931348623157e+308  
console.log(Number.MAX\_VALUE > 1e308); // Output: true

## 10. Number.MIN\_VALUE

- **Syntax:** Number.MIN\_VALUE
- **Description:** Represents the smallest positive numeric value that can be represented in JavaScript.
- **Example:** console.log(Number.MIN\_VALUE); // Output: 5e-324  
console.log(Number.MIN\_VALUE < 1e-323); // Output: true

## 11. Number.POSITIVE\_INFINITY

- **Syntax:** Number.POSITIVE\_INFINITY
- **Description:** Represents positive infinity.
- **Example:**  
console.log(Number.POSITIVE\_INFINITY); // Output: Infinity  
console.log(1 / 0); // Output: Infinity

## 12. Number.NEGATIVE\_INFINITY

- **Syntax:** Number.NEGATIVE\_INFINITY
- **Description:** Represents negative infinity.
- **Example:** console.log(Number.NEGATIVE\_INFINITY); // Output: -Infinity  
console.log(-1 / 0); // Output: -Infinity



# **BOOLEAN**

- JavaScript Boolean is an object that represents value in two states: true or false.
- JavaScript Boolean Properties

## **Property      Description**

|             |                                                                        |
|-------------|------------------------------------------------------------------------|
| constructor | returns the reference of Boolean function that created Boolean object. |
| prototype   | enables you to add properties and methods in Boolean prototype.        |

## **❖ Boolean Methods:**

- Method      Description
- toSource()      returns the source of Boolean object as a string.
- toString()      converts Boolean into String.
- valueOf()      converts other type into Boolean.

### **1. Boolean.prototype.toSource()**

- **Syntax:** booleanInstance.toSource()
- **Description:** Returns a string representing the source code of the Boolean object. This method is non-standard and not available in all JavaScript environments.
- **Example:**

```
let bool = new Boolean(true);
console.log(bool.toSource()); // Output: (new Boolean(true))
```

**Note:** The toSource() method is non-standard and primarily available in Firefox. It is not recommended for use in production code.

### **2. Boolean.prototype.toString()**

- **Syntax:** booleanInstance.toString()
- **Description:** Converts a Boolean value to a string ("true" or "false").
- **Example:**

```
let boolTrue = true;
let boolFalse = false;
console.log(boolTrue.toString()); // Output: 'true'
console.log(boolFalse.toString()); // Output: 'false'
```

### **3. Boolean.prototype.valueOf()**

- **Syntax:** booleanInstance.valueOf()
- **Description:** Returns the primitive value of a Boolean object, effectively converting other types into Boolean.
- **Example:**

```
let boolObjectTrue = new Boolean(true);
let boolObjectFalse = new Boolean(false);
console.log(boolObjectTrue.valueOf()); // Output: true
console.log(boolObjectFalse.valueOf()); // Output: false
// Converting other types to Boolean
console.log(Boolean(1).valueOf()); // Output: true
console.log(Boolean(0).valueOf()); // Output: false
console.log(Boolean("hello").valueOf()); // Output: true
console.log(Boolean("").valueOf()); // Output: false
```

# OOPS CONCEPTS

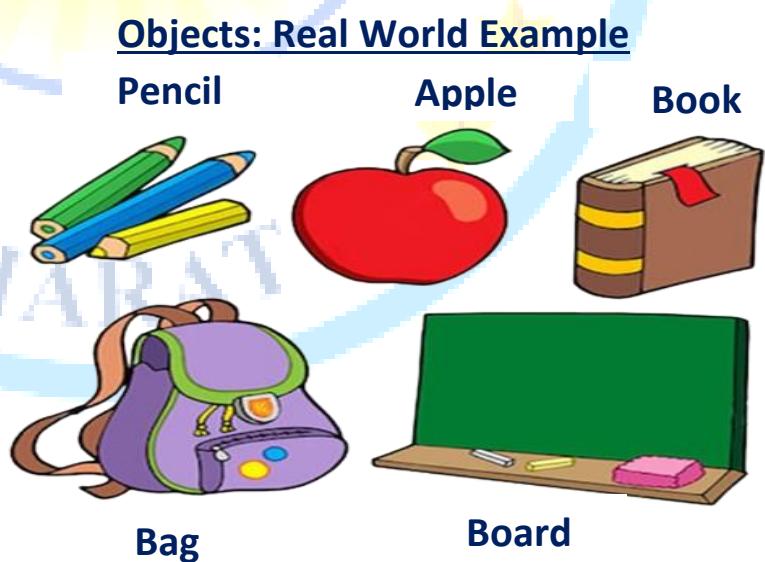
## Object Oriented Programming System

- Object-Oriented Programming is a methodology or paradigm to design a program using **classes** and **objects**.
- The programming paradigm where everything is represented as an **object** is known as truly **object-oriented programming** language. Smalltalk is considered as the first truly object-oriented programming language.
- It simplifies the software development and maintenance by providing following concepts.
  1. Object
  2. Class
  3. Inheritance
  4. Polymorphism
  5. Abstraction
  6. Encapsulation

### 1. Object:

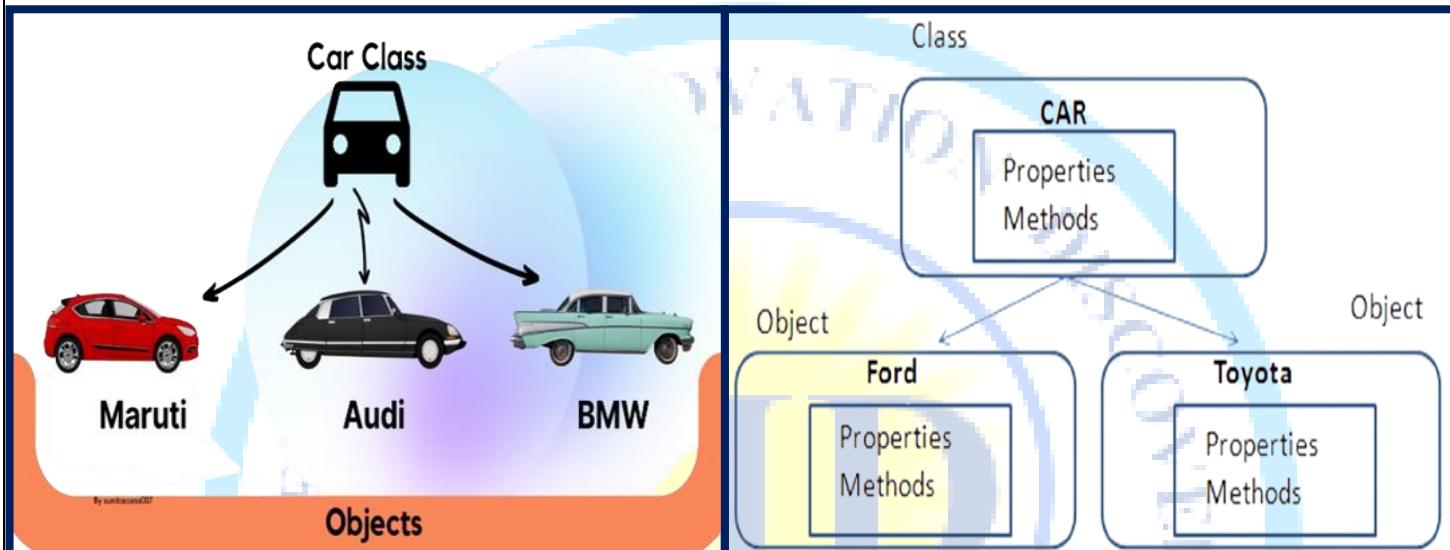
- Object means a real world entity such as pen, chair, table etc.
- Any entity that has **state** and **behavior** is known as an object.
- For example: chair, pen, table, keyboard, bike etc. It can be **physical** and **logical**.
- **State** tells us how the object looks or what properties it has. **Behavior** tells us what the object does.
- object is a specific **instance** of a class.
- **instance** is a specific realization of any objects.

#### Example:-



## 2. Class:

- Collection of **objects** is called class. It is a logical entity.
- To create objects, we required some model or plan or template or blue print, which is nothing but class.
- We can write a class to represent **properties(attributes)** and **actions (behaviour)** of object.
- **Properties** can be represented by **variable**
- **Action** can be represented by **methods**.
- Hence class contains both **variables** and **method**.
- a class is a template definition of the methods and variables in a particular kind of object. Thus, an object is a specific **instance of a class**.



# Class                      Object

Create an instance

Car

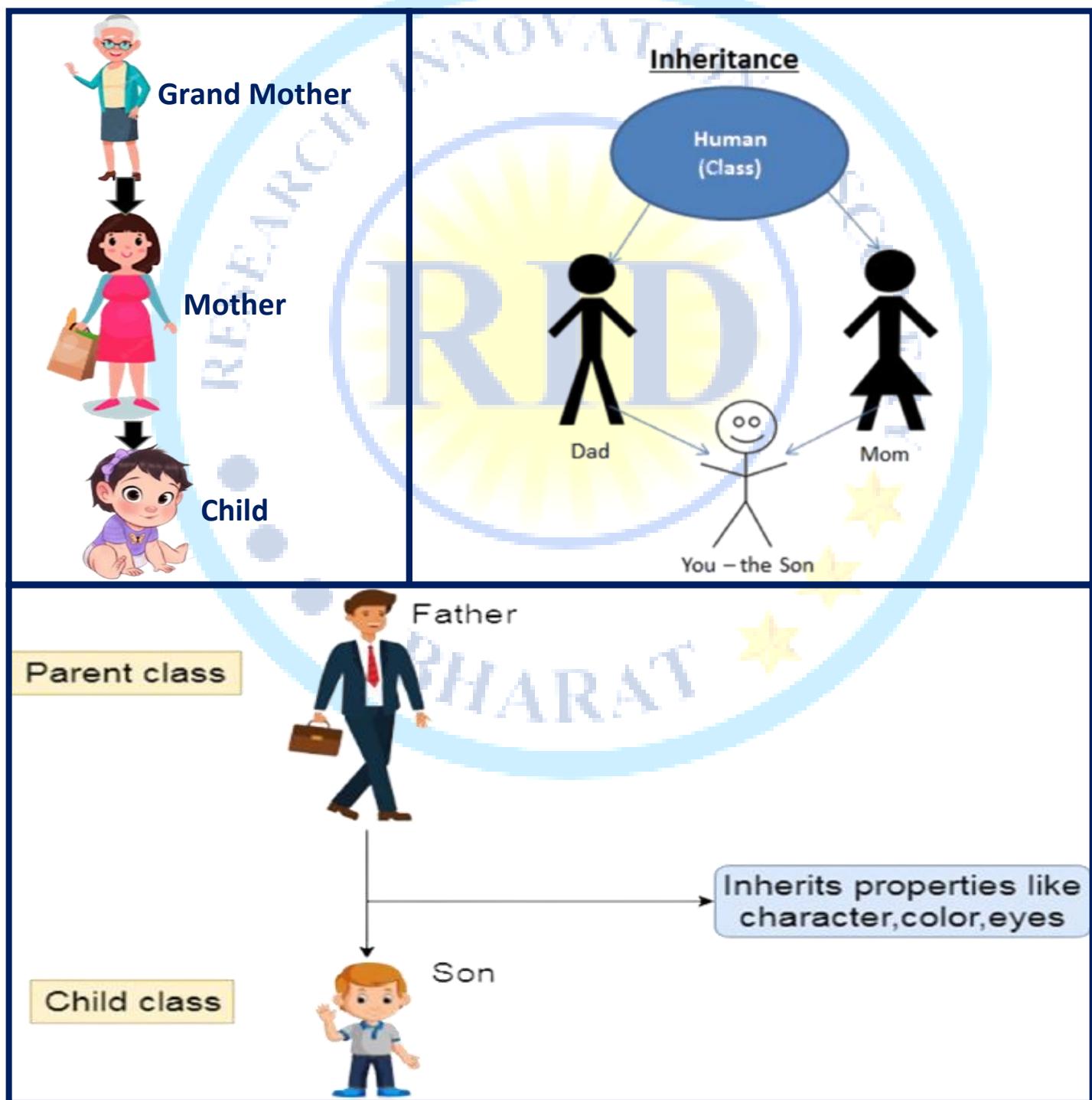


| Properties | Methods - behaviors |
|------------|---------------------|
| color      | start()             |
| price      | backward()          |
| km         | forward()           |
| model      | stop()              |

| Property values | Methods    |
|-----------------|------------|
| color: red      | start()    |
| price: 23,000   | backward() |
| km: 1,200       | forward()  |
| model: Audi     | stop()     |

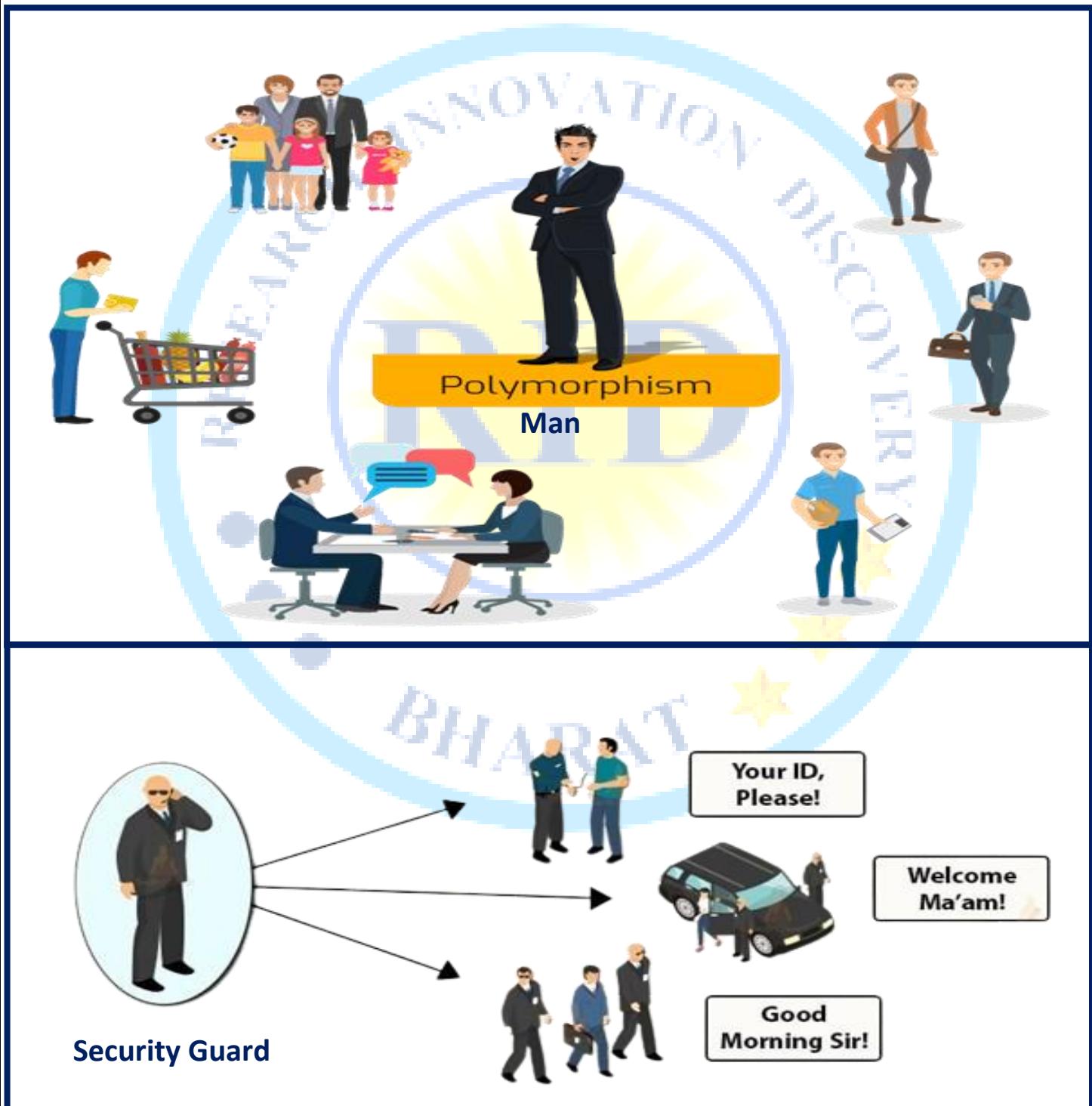
### 3. Inheritance:

- When one object acquires all the **properties** and **behaviours** of **parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.
- **Sub class** - Subclass or Derived Class refers to a class that receives properties from another class.
- **Super class** - The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.
- **Reusability** - when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class that is inheritance.



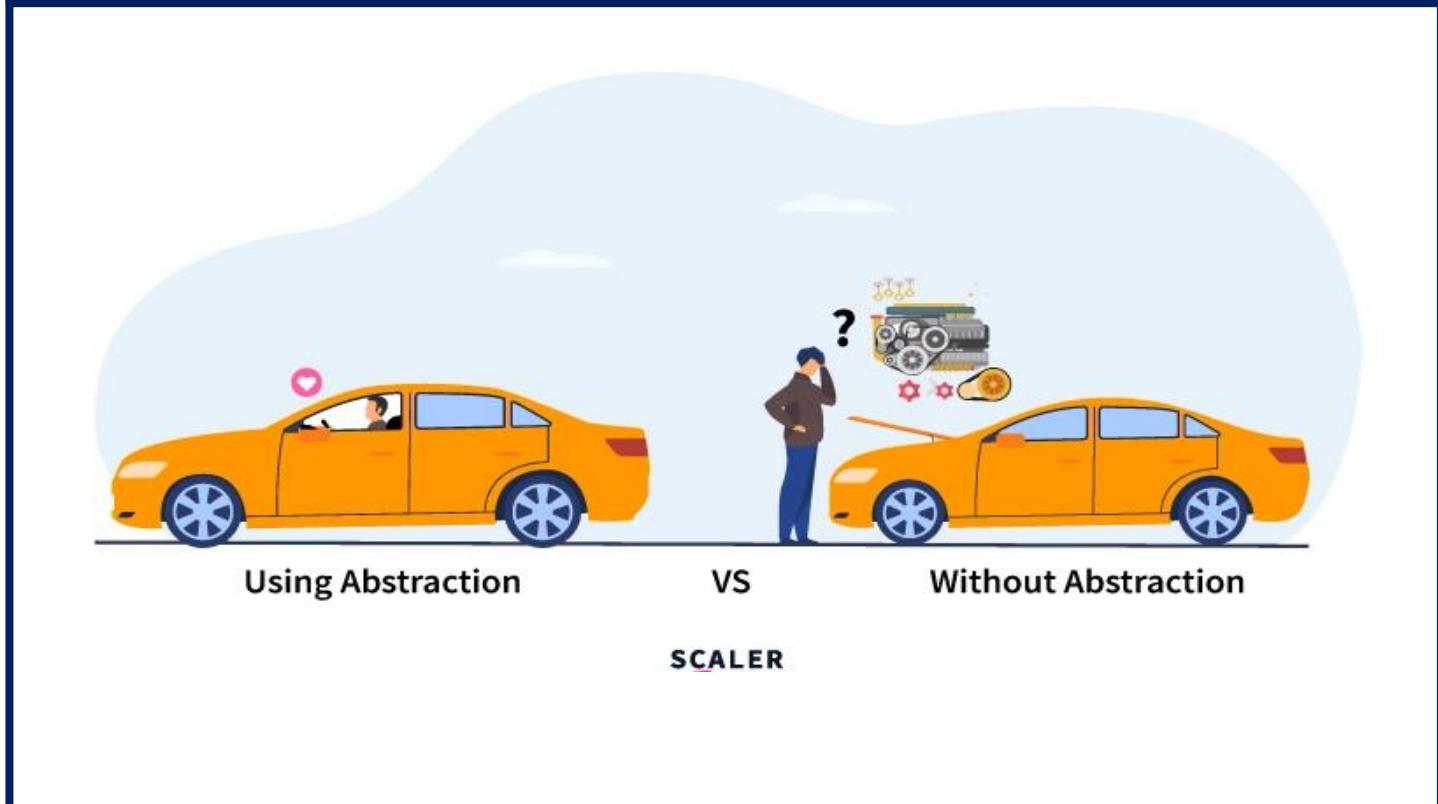
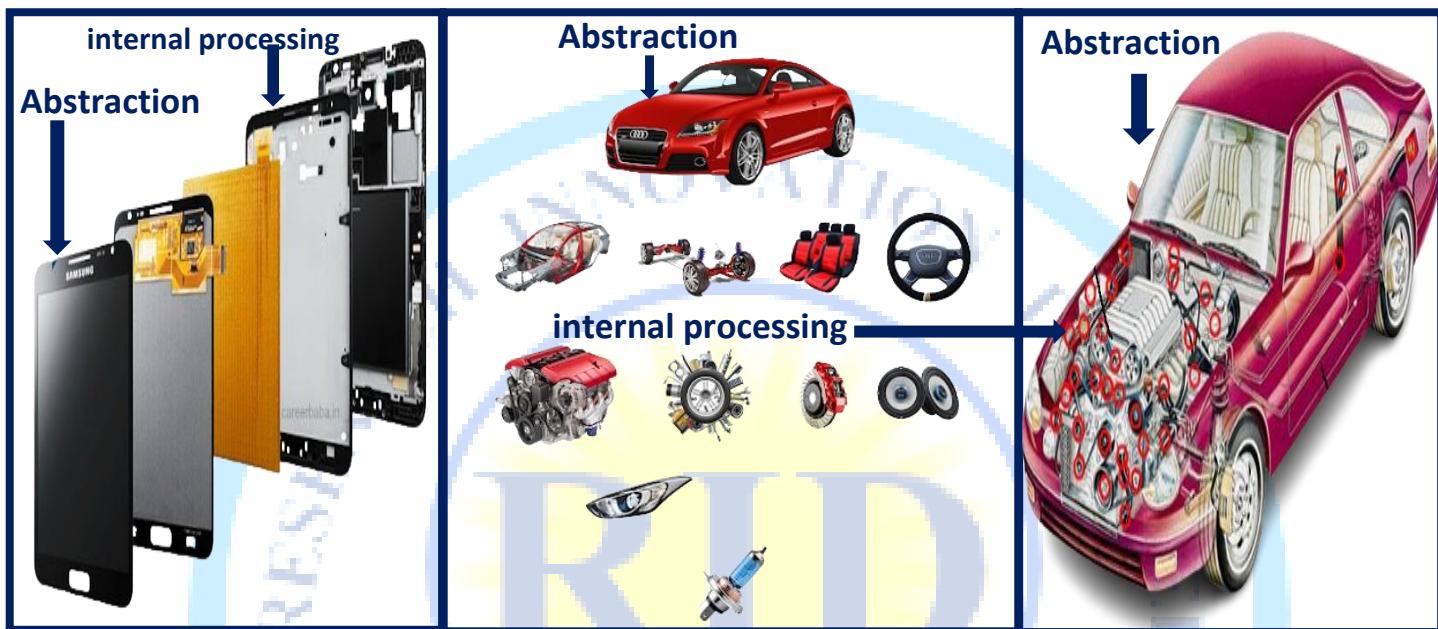
## 1. Polymorphism:

- When one task is performed by different ways i.e. known as polymorphism.
- Polymorphism means "many forms"
- Imagine a man named Sangam. Sangam has different roles depending on the context:
  - ✓ At work, Sangam is an Engineer.
  - ✓ At home, Sangam is a father.
  - ✓ In a social gathering, Sangam is a Friend.
  - ✓ In flight, Sangam is passenger.



## 2. Abstraction:

- Hiding **internal** details and showing **functionality** is known as abstraction.
- Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information. For example: phone call and Car. we don't know the internal processing.
- In C++, we use abstract class and interface to achieve abstraction.



### 3. Encapsulation:

- Binding (or wrapping) **code** and **data** together into a single unit is known as encapsulation.
- encapsulation is described as the process of combining code (**methods or functions**) and data (**variables or attributes**) into a single unit.
- **Methods** or functions represent **executable** code, while data or variables represent **stored** information within a program.

The collage consists of five panels:

- Top Left:** A blue capsule labeled "Methods" and a red capsule labeled "Variable" are shown merging into a single blue capsule. A blue arrow points from the word "class" to the first capsule.
- Top Right:** A blue capsule labeled "Encapsulation" contains a colorful fruit mix. Arrows point from the capsule to three boxes: "Class", "Methods", and "Variables".
- Middle Left:** A code snippet for a class definition is shown: "class { data (variables) + methods }". A bracket groups "data (variables)" and "methods", with the word "encapsulation" written next to it.
- Middle Right:** A car's engine compartment is shown. Labels point to the "engine (class)", "variables (gear)", and "(battery) methods".
- Bottom Left:** A smartphone screen displays various app icons. A bracket groups the screen and the internal circuit board, with "Encapsulation" above and "Abstraction" below.
- Bottom Right:** A calculator screen shows the result "2673.42" of the equation "18923+891.12". A bracket groups the calculator and its internal circuit board, with "Encapsulation" above and "Abstraction" below. Two callout boxes provide detailed explanations:
  - Encapsulation:** Explains that for a mathematical equation, the calculator hides the complex calculation steps (implementation) and only shows the result (final value).
  - Abstraction:** Explains that the user doesn't need to know how the calculator's battery module works to use it.
 A purple box at the bottom states: "Using Battery module along with other modules we use calculator -> Thus using Abstraction encapsulation is performed".

## **ADVANTAGES OF OOPS**

- 1) We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
- 2) OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
- 3) The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
- 4) OOP systems can be easily upgraded from small to large systems.
- 5) It is possible that multiple instances of objects co-exist without any interference,
- 6) It is very easy to partition the work in a project based on objects.
- 7) It is possible to map the objects in problem domain to those in the program.
- 8) The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- 9) By using inheritance, we can eliminate redundant code and extend the use of existing classes.
- 10) Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- 11) The data-centered design approach enables us to capture more details of model in an implementable form.

## **Disadvantages of OOPS**

- 1) The length of the programmes developed using OOP language is much larger than the procedural approach. Since the programme becomes larger in size, it requires more time to be executed that leads to slower execution of the programme.
- 2) We can not apply OOP everywhere as it is not a universal language. It is applied only when it is required. It is not suitable for all types of problems.
- 3) Programmers need to have brilliant designing skill and programming skill along with proper planning because using OOP is little bit tricky.
- 4) OOPs take time to get used to it. The thought process involved in object-oriented programming may not be natural for some people.
- 5) Everything is treated as object in OOP so before applying it we need to have excellent thinking in terms of objects.

# **OBJECT**

**Object:-** 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 etc.
- An Object is an instance of a class

## **❖ How to Create the object in JavaScript?**

- The object can be created in two ways in JavaScript:
  1. Object Literal
  2. Object Constructor

### **1. Object Literal:-**

- an object literal is a way to create an object using a simple and concise syntax. An object in JavaScript is a collection of key-value pairs, where each key is a string (or symbol) and its value can be any data type (such as a string, number, function, or another object).
- **Syntax of an Object Literal**
  - An object literal is defined by enclosing a comma-separated list of key-value pairs within curly braces {}.

#### **Example:**

```
let person={  
    name:"Sangam Kumar",  
    age:28,  
    contact:900500506,  
    getDetails: function(){  
        console.log(`Name is ${this.name}\nAge is ${this.age}\nContact is- ${this.contact}`)  
    }  
}  
console.log(person)  
person.getDetails()
```

#### **Output:**

```
{  
    name: 'Sangam Kumar',  
    age: 28,  
    contact: 900500506,  
    getDetails: [Function: getDetails]  
}  
Name is Sangam Kumar  
Age is 28  
Contact is- 900500506
```



## **2. Object Constructor:**

- object constructor is a function used to create and initialize objects with a specific structure and properties. It acts as a blueprint for creating multiple objects with the same characteristics.

**Example:**

### **1. Using the Built-In Object() Constructor**

- You can create a new object using the built-in Object() constructor function:

**Ex.**

```
let obj = new Object();
```

- This creates an empty object, which is essentially same as using the object literal {}. However, using the Object() constructor is less common since the object literal syntax is more concise.

## **2. Custom Object Constructors**

- A more powerful use of the Object Constructor is to create custom constructors. This allows you to define a template for creating objects with specific properties and methods.

**Example:** Custom Object Constructor

```
function Person(firstName, lastName, age) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.age = age;  
    this.greet = function() {  
        console.log("Hello, " + this.firstName);  
    };  
}
```

```
let person1 = new Person("John", "Doe", 30);  
let person2 = new Person("Jane", "Smith", 25);  
  
console.log(person1.firstName); // Output: John  
console.log(person2.age); // Output: 25  
  
person1.greet(); // Output: Hello, John
```

**Example:**

```
function Person(name,age,address){  
    this.name=name  
    this.age=age  
    this.address=address  
}  
let p1 = new Person("Jai Kumar",30,"Patna")  
console.log("Name:", p1.name, "\n", "Age:", p1.age, "\n", "Address:", p1.address)
```

**Output:**

```
Name: Jai Kumar  
Age: 30  
Address: Patna
```

## ❖ Declaring class in ES6:

### Example-1:

```
class Person{  
    constructor(name,age,address){  
        this.name=name  
        this.age=age  
        this.address=address  
    }  
    getDetails(){  
        console.log(`Name is ${this.name}\nAge is ${this.age}\nContact is- ${this.address}`)  
    } } //here Person has got 3 attribute- name,age,address& one method- getDetails()  
let ob1 = new Person("Sangam Kumar",29,"Bihata Patna")//here object is created using new  
keyword. hence ob1 is object & Person is class  
ob1.getDetails()
```

**Output:** Name is Sangam Kumar

Age is 29  
Contact is- Bihata Patna  
Create a user choice based Calculator using OOP

### Example-2: class Calculator{

```
    constructor(num1,num2){  
        this.num1=num1  
        this.num2=num2 }  
    add(){ return this.num1+this.num2 }  
    sub(){ return this.num1-this.num2 }  
    mul(){ return this.num1*this.num2 }  
    div(){ return this.num1/this.num2 }  
} //DYNAMIC INPUT IN node js //Install the module prompt-sync // npm i prompt-sync  
const prompt = require("prompt-sync")();  
let n1=parselnt(prompt("Enter number1- "))  
let n2=parselnt(prompt("Enter number2- "))  
let cl=new Calculator(n1,n2)  
console.log("Choose 1 for Addition\nChoose 2 for Subtraction\nChoose 3 for  
Multiplication\nChoose 4 for Division")  
let ch=parselnt(prompt("Enter your choice: "))  
if(ch==1){ console.log( "Sum=", cl.add() ) }  
else if(ch==2){ console.log("Substration=", cl.sub() ) }  
else if(ch==3){ console.log("Multiplication=", cl.mul() ) }  
else if(ch==4){ console.log("Division=", cl.div() ) }  
else{ console.log("Invalid choice") }
```

**Output:** Enter number1- 10

Enter number2- 20  
Choose 1 for Addition  
Choose 2 for Subtraction  
Choose 3 for Multiplication  
Choose 4 for Division  
Enter your choice: 3  
Multiplication= 200

## ❖ **INHERITANCE**

- Acquire properties and methods from another object. This is a fundamental concept in object-oriented programming (OOP) and enables the creation of hierarchical relationships between objects, promoting code reuse and organization.

### ➤ Key Concepts of Inheritance in JavaScript:

#### 1. Prototype-Based Inheritance:

- JavaScript uses prototype-based inheritance, meaning every object has a prototype object from which it can inherit properties and methods. object itself. If it doesn't find it, it looks at object's prototype, and so on, up prototype chain. cvx

#### 2. Prototype Chain:

- The chain of prototypes an object can follow is called the prototype chain. At the top of this chain is Object.prototype, which provides default methods like `toString()` and `valueOf()`.
- If a property is not found in any prototype in the chain, `undefined` is returned.

#### Example: Inheritance Using Prototypes

```
function Animal(name) {  
    this.name = name;  
}  
Animal.prototype.speak = function() {  
    console.log(` ${this.name} makes a sound.`);  
};  
function Dog(name, breed) {  
    Animal.call(this, name); // Call the Animal constructor with the Dog's name  
    this.breed = breed;  
} // Inherit methods from Animal's prototype  
Dog.prototype = Object.create(Animal.prototype);  
Dog.prototype.constructor = Dog;  
Dog.prototype.bark = function() {  
    console.log(` ${this.name} barks.`);  
};  
let myDog = new Dog("Buddy", "Golden Retriever");  
myDog.speak(); // Output: Buddy makes a sound.  
myDog.bark(); // Output: Buddy barks.
```

#### Explaiiton:

This code demonstrates **prototypal inheritance** in JavaScript, where a Dog class inherits from an Animal class. Let's break it down step by step:

#### 1. Define the Animal Constructor

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

- This is a constructor function that initializes an Animal object with a name property.
- `this.name` refers to the name passed when creating a new instance of Animal.

#### 2. Add a Method to Animal Prototype

```
Animal.prototype.speak = function() {  
    console.log(` ${this.name} makes a sound.`);  
};
```



- speak is a method added to the Animal prototype.
- It allows all instances of Animal to use this method.
- this.name refers to the specific instance's name property.

### 3. Define the Dog Constructor

```
function Dog(name, breed) {
    Animal.call(this, name); // Call Animal constructor
    this.breed = breed;
}
```

- Dog is another constructor function for creating a Dog object.
- Animal.call(this, name):
  - Calls the Animal constructor with this bound to the Dog instance.
  - This initializes name property in the Dog instance using the logic from Animal constructor.
- this.breed adds a breed property specific to Dog.

### 4. Set Up Inheritance

- ```
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
```
- Dog.prototype = Object.create(Animal.prototype):
    - Creates a new object (Dog.prototype) with its prototype set to Animal.prototype.
    - This ensures Dog inherits all methods from Animal.
  - Dog.prototype.constructor = Dog:
    - Resets the constructor property of Dog.prototype to Dog (otherwise, it would point to Animal).

### 5. Add a New Method to Dog Prototype

```
Dog.prototype.bark = function() {
    console.log(`#${this.name} barks.`);
};
```

- Adds a bark method specific to Dog instances.

### 6. Create an Instance of Dog

```
let myDog = new Dog("Buddy", "Golden Retriever");
```

- Creates a new Dog object.
- name is set to "Buddy" and breed is set to "Golden Retriever".
- Since Dog inherits from Animal, it also gets access to methods defined on Animal.prototype.

### 7. Use the Methods

- ```
myDog.speak(); // Output: Buddy makes a sound.
myDog.bark(); // Output: Buddy barks.
```
- myDog.speak() calls the speak method from Animal.prototype.
  - myDog.bark() calls the bark method from Dog.prototype.

---

#### Key Points

1. **Inheritance:** Dog inherits properties and methods from Animal through Object.create.
2. **Encapsulation:** Shared methods are added to prototypes instead of individual instances to save memory.
3. **Constructor Linking:** Animal.call(this, name) ensures Dog instances initialize name like Animal.

## TYPES OF INHERITANCE

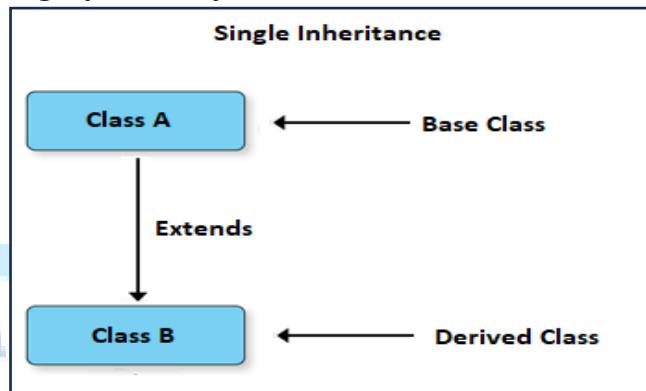
### ❖ 1. Single Inheritance:

- When a child object or class inherits from a single parent object or class.

Example:

### 1. Define the Father Class (Parent Class)

```
class father{
    constructor(fname){
        this.fname=fname
    }
    print(){
        console.log("Father name is ", this.fname)
    }
} // let obj=new father("sangam kumar")
// obj.print()// Output: Father name is Sangam Kumar
```



### 2. Define the Son Class (Child Class)

```
class son extends father{
    // Son class: Inherits from the Father class using the extends keyword.
    constructor(fname, sname){
        super(fname) // Uses super keyword to call the parent class's constructor, passing the
        fname argument.
        this.sname=sname
    }
    print(){
        console.log("Father's Name is",this.fname)
        console.log("Child name is ", this.sname)
    }
}
```

### 3. Create an Instance of Son and Call the print() Method

```
let obj2=new son("Mohan Kumar", "Sohan Kumar")
obj2.print()
```

#### Output:

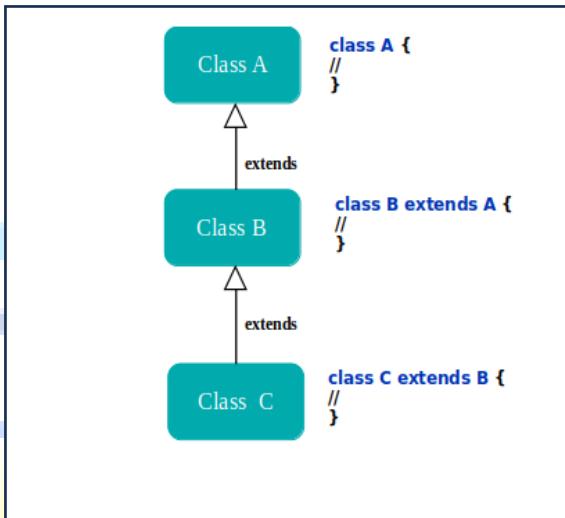
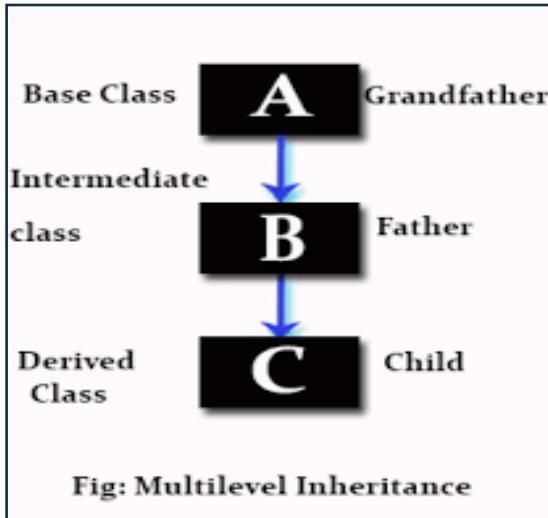
Father's Name is Mohan Kumar  
Child name is Sohan Kumar

#### Key Concepts Demonstrated

- Class-based Inheritance:**
  - The Son class inherits properties and methods from the Father class using the extends keyword.
- super Keyword:**
  - Used in the Son class constructor to call the parent class (Father) constructor and initialize properties (fname).
- Method Overriding:**
  - The Son class overrides the print() method of the Father class to provide additional functionality.
- Polymorphism:**
  - Both Father and Son have a print() method, but the behavior changes depending on the class instance.

## ❖ 2. Multilevel Inheritance:

- ❖ **Definition:** A type of inheritance where a child class inherits from a parent class, and then a subclass inherits from that child class. This forms a chain of inheritance.
- ❖ **Supported in JavaScript:** Yes, JavaScript supports multilevel inheritance using class and prototype.



### Example: 1. GrandFather Class (Base Class)

```

class grandfather{
    constructor(gname){
        this.gname=gname
    }
    disp(){
        console.log("Grandfather name is",this.gname)
    }
} 
```

### 2. Father Class (Child of GrandFather)

```

class Father extends grandfather{
    constructor(gname,fname){
        super(gname)
        this.fname=fname
    }
    disp(){
        console.log("Grandfather name is ", this.gname)
        console.log("Fahter name is ",this.fname)
    }
} 
```

### 3. Son Class (Child of Father)

```

class son extends Father{
    constructor(gname,fname,sname){
        super(gname,fname)
        this.sname=sname
    }
    disp(){
        console.log("GrandFather name: ", this.gname);
        console.log("Father name: ", this.fname);
        console.log("Son's name:", this.sname);
    }
}
let obj1=new son("Mohan","Sohan","Johan")
obj1.disp() 
```

### Key Concepts Demonstrated

1. **Multilevel Inheritance:**
  - o The Son class inherits from the Father class, which in turn inherits from the GrandFather class.
  - o This forms a chain: GrandFather → Father → Son.
2. **super Keyword:**
  - o Used in the constructors of Father and Son to call the parent class constructor and initialize inherited properties.
3. **Method Overriding:**
  - o Each class defines its own disp() method, which overrides the disp() method of its parent class to display more specific information.
4. **Polymorphism:**
  - o Even though all three classes have a disp() method, the behavior changes depending on the instance.

### Output:

GrandFather name: Mohan  
Father name: Sohan  
Son's name: Johan

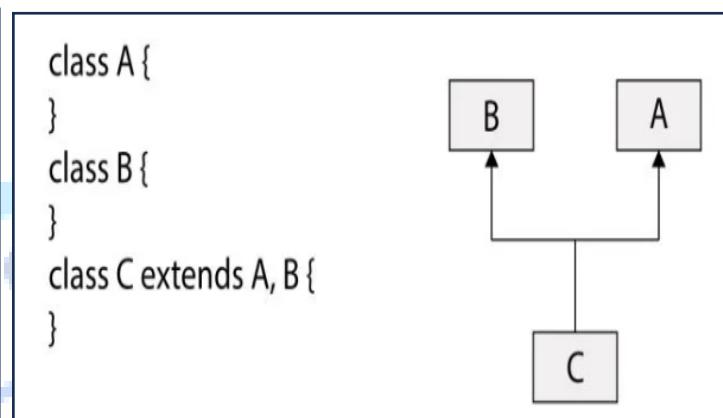
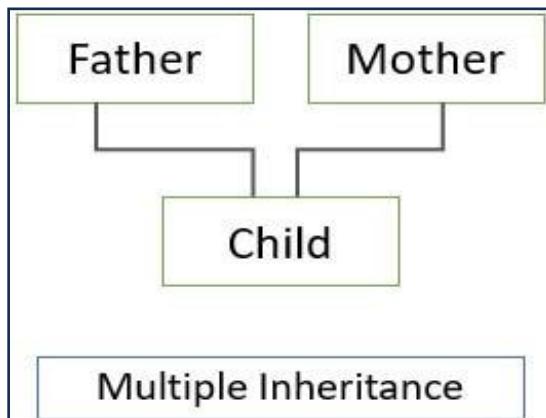
### Output:

For ob1 (instance of Father):  
GrandFather name: Mohan  
Father name: Sohan



### ❖ 3. Multiple Inheritance:

- **Definition:** A type of inheritance where a class can inherit from multiple parent classes at the same time.
- **Supported in JavaScript:** No direct support. However, it can be achieved through **mixins** or **composition**.

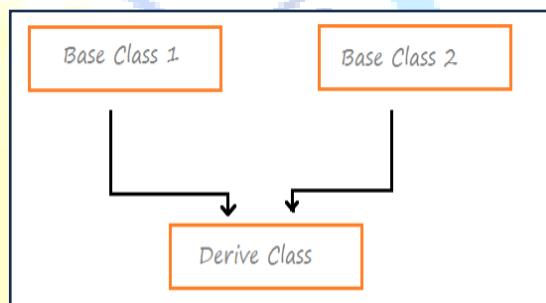


#### Example-1: 1. Defining the Mixins

```

let flyingMixin = {
  fly() {
    console.log(`${this.name} is flying!`);
  }
};
let swimmingMixin = {
  swim() {
    console.log(`${this.name} is swimming!`);
  }
}; // Mixins are plain objects that contain methods (and sometimes properties) that can be "mixed" into other classes.

```



#### 2. Creating the Bird Class

```

class Bird {
  constructor(name) {
    this.name = name;
  } // The Bird class represents a generic bird.
// It has a constructor that initializes the name property, which is specific to each bird instance.

```

#### 3. Adding Behaviors to the Bird Class

```

Object.assign(Bird.prototype, flyingMixin, swimmingMixin);
// Object.assign(target, ...sources):
// Copies the properties and methods from one or more source objects (flyingMixin and swimmingMixin) into the target object (Bird.prototype).
// The Bird class prototype now has the fly() and swim() methods, enabling instances of Bird to use these methods.

```

#### 4. Creating an Instance of the Bird Class

```
let duck = new Bird("Duck"); // A new instance of the Bird class is created with the name "Duck".
```

#### 5. Calling the Mixed-in Methods

```

duck.fly(); // Output: Duck is flying!
duck.swim(); // Output: Duck is swimming!

```

#### Example-2: // Define mixins

```
let canRunMixin = {
  run() {
    console.log(`${this.name} is running!`);
  }
};

let canJumpMixin = {
  jump() {
    console.log(`${this.name} is jumping!`);
  }
}; // Define a base class

class Animal {
  constructor(name) {
    this.name = name;
  } // Add mixins to the Animal class
}

Object.assign(Animal.prototype, canRunMixin, canJumpMixin);

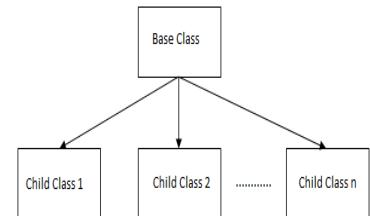
// Create an instance of Animal
let rabbit = new Animal("Rabbit"); // Call the mixed-in methods
rabbit.run(); // Output: Rabbit is running!
rabbit.jump(); // Output: Rabbit is jumping!
```

## 4. Hierarchical Inheritance

- When multiple child objects inherit from the same parent object

#### Example: 1. The Animal Class (Parent Class)

```
class Animal {
  speak() {
    console.log("Animal makes a sound.");
  }
}
```



**Purpose:** This method is intended to represent a general behavior common to all animals.

#### 2. The Dog Class (Child of Animal)

```
class Dog extends Animal {
  bark() {
    console.log("Dog barks.");
  }
}
```

// Inheritance: The Dog class extends the Animal class using the extends keyword.

// Purpose: Represents a specific type of animal (Dog) with unique behavior (bark())

#### The Cat Class (Child of Animal)

```
class Cat extends Animal {
  meow() {
    console.log("Cat meows.");
  }
}
```

// Inheritance: The Cat class also extends the Animal class. **Purpose:** Represents another specific type of animal (Cat) with unique behavior (meow()).

#### 4. Creating Instances and Calling Methods

```
let myDog = new Dog();
let myCat = new Cat();
```

#### 5. Calling the speak() Method

```
myDog.speak(); // Output: Animal makes a sound.
myCat.speak(); // Output: Animal makes a sound.
```

## ❖ Polymorphism:

- Polymorphism in JavaScript refers to the ability of different objects to respond to the same method or function call in their own way.

### 1. Polymorphism with Classes and Method Overriding

- When a subclass overrides a method from its parent class to provide a specific implementation.

#### 1.Method overriding---inheritance is mandatory

**Example:**

```
class Animal {  
    speak() {  
        console.log("The animal makes a sound.");  
    }  
}  
class Dog extends Animal {  
    speak() {  
        console.log("The dog barks.");  
    }  
}  
class Cat extends Animal {  
    speak() {  
        console.log("The cat meows.");  
    }  
}  
const animals = [new Animal(), new Dog(), new Cat()];  
animals.forEach(animal => animal.speak());
```

**Output:**

The animal makes a sound.  
The dog barks.  
The cat meows.

### 2. Polymorphism with Interfaces (Duck Typing)

- Different objects can implement the same method and behave differently.

**Example:**

```
class Circle {  
    draw() {  
        console.log("Drawing a Circle.");  
    }  
}  
class Square {  
    draw() {  
        console.log("Drawing a Square.");  
    }  
}  
class Triangle {  
    draw() {  
        console.log("Drawing a Triangle.");  
    }  
}  
const shapes = [new Circle(), new Square(), new Triangle()];  
shapes.forEach(shape => shape.draw());
```

**Output:**

Drawing a Circle.  
Drawing a Square.  
Drawing a Triangle.

### ❖ Function Overloading:

- JavaScript does not support function overloading natively (like in some other languages), but it can be simulated using default parameters or by checking the arguments.

#### **Example-1: by checking the arguments**

```
function greet(name, age) {  
    if (age !== undefined) {  
        console.log(`Hello ${name}, you are ${age} years  
old. `);  
    } else {  
        console.log(`Hello ${name}.`);  
    }  
}  
  
// Calling the function with different arguments  
greet("Sangam"); // Output: Hello Sangam.  
greet("Bob", 25); // Output: Hello Bob, you are  
25 years old.
```

#### **Example-1: Using Default Arguments**

```
function greet(name, age = null) {  
    if (age !== null) {  
        console.log(`Hello ${name}, you are ${age} years old.`);  
    } else {  
        console.log(`Hello ${name}.`);  
    }  
  
    // Calling the function with different arguments  
    greet("Alice"); // Output: Hello Alice.  
    greet("Bob", 25); // Output: Hello Bob, you are 25  
    years old.
```

### ❖ Operator Overloading:

JavaScript does not allow direct operator overloading, but you can achieve a similar effect by overriding behavior of object methods like `toString` or using `valueOf` method for custom operations.

#### **Example: Overloading the + Operator**

```
class ComplexNumber {  
    constructor(real, imaginary) {  
        this.real = real;  
        this.imaginary = imaginary;  
    }  
    // Custom addition for complex numbers  
    add(other) {  
        return new ComplexNumber(  
            this.real + other.real,  
            this.imaginary + other.imaginary  
        );  
    }  
    toString() {  
        return `${this.real} + ${this.imaginary}i`;  
    }  
}  
const num1 = new ComplexNumber(3, 4);  
const num2 = new ComplexNumber(1, 2);  
const result = num1.add(num2);  
console.log(result.toString()); // Output: 4 + 6i
```

### Key Notes:

1. **Function Overloading:** Simulated using conditions or default arguments.
2. **Operator Overloading:** Achieved by defining custom methods (e.g., `add` or overriding `toString`/`valueOf`).

## ❖ Abstraction

- Abstraction is the concept of hiding implementation details and showing only the essential features of an object or function. In JavaScript, abstraction can be achieved using:
  - Classes and Methods:** Encapsulating implementation details inside methods.
  - Closures:** Using private variables that are accessible only through specific functions.

### Comparison with Public Methods

| Aspect       | Private Method ( #igniteEngine )  | Public Method ( start )         |
|--------------|-----------------------------------|---------------------------------|
| Access Scope | Only accessible within the class. | Accessible from anywhere.       |
| Purpose      | Internal implementation detail.   | Public interface for the class. |
| Syntax       | Prefixed with # .                 | No prefix required.             |

#### Example 1: Abstraction with Classes

- Using methods to hide complex implementation details and expose only what's necessary.

```
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
    }
    // Public method to start the car
    start() {
        this.#raj();
        console.log(` ${this.make} ${this.model} is starting...`);
    } // Private method (implementation detail)
    #raj() {
        console.log("Igniting the engine...");
    }
}
const myCar = new Car("Toyota", "TATA");
myCar.start(); // Output: Igniting the engine... \n Toyota TATA is starting...
// myCar.#raj(); // Error: Private field '#igniteEngine' must be declared in an enclosing class
```

#### Example-2: Student Class with Abstraction

```
class Student {
    constructor(name, rollNumber) {
        this.name = name; // Public property
        this.rollNumber = rollNumber; // Public property
    }
    // Public method to display student details and final grade
    displayDetails() {
        const grade = this.#calculateGrade();
        console.log(`Student Name: ${this.name}`);
        console.log(`Roll Number: ${this.rollNumber}`);
        console.log(`Final Grade: ${grade}`);
    }
}
```



```
// Private method (implementation detail)
#calculateGrade() {
    // Simulate complex grade calculation logic
    console.log("Calculating grade based on performance...");
    return "A"; // Simplified grade logic for demonstration
} } // Using the Student class
const student1 = new Student("Ankit Kumar", 101);
student1.displayDetails();
```

**Output:**

```
Calculating grade based on performance...
```

```
Student Name: Ankit Kumar
```

```
Roll Number: 101
```

```
Final Grade: A
```

```
// student1.#calculateGrade(); // Error: Private field '#calculateGrade' must be declared in an
enclosing class
```

### **Example 3: Abstraction with Closures**

Using closures to create private variables and methods.

```
function createCounter() {
    let count = 0; // Private variable
    return {
        increment() {
            count++;
            console.log(`Count: ${count}`);
        },
        decrement() {
            count--;
            console.log(`Count: ${count}`);
        },
        getCount() {
            return count; // Exposing the value through a method
        }
    };
}
const counter = createCounter();
counter.increment(); // Output: Count: 1
counter.increment(); // Output: Count: 2
console.log(counter.getCount()); // Output: 2
// console.log(counter.count); // Undefined (private variable)
```

#### **Key Benefits of Abstraction:**

1. Encapsulation: Hides unnecessary details, exposing only relevant features.
2. Modularity: Makes the code easier to manage and reuse.
3. Security: Prevents unauthorized access to sensitive data.

```
if (this.marks >= 90 && this.marks <= 100) { return "A"; } else if (this.marks >= 80 && this.marks < 90) { return "B"; } else if (this.marks >= 70 && this.marks < 80) { return "C"; } else if (this.marks >= 60 && this.marks < 70) { return "D"; } else if (this.marks < 60) { return "Fail"; } else { return "Invalid Marks"; }
```



## ❖ Encapsulation:

it is the practice of bundling data (properties) and methods (functions) into a single unit, typically a class, and restricting direct access to some of the object's components.

### Key Features of Encapsulation

#### 1. Data Hiding:

- Internal details of an object (e.g., private variables) are hidden from the outside.
- Access is controlled through getter and setter methods or private fields.

#### 2. Controlled Access:

- Public methods provide controlled access to the hidden data.
- Prevents unintended interference or misuse.

#### 3. Improved Maintainability:

- Internal logic can be changed without affecting external code that interacts with the object.

### Example: Bank Account with Encapsulation

```
class BankAccount {  
    constructor(owner, balance) {  
        this.owner = owner; // Public property  
        this.#balance = balance; // Private property  
    } // Getter for balance (read-only access)  
    getBalance() {  
        return `The balance for ${this.owner} is $$ ${this.#balance}.`;  
    } // Method to deposit money (controlled modification)  
    deposit(amount) {  
        if (amount > 0) {  
            this.#balance += amount;  
            console.log(`$$ ${amount} deposited successfully.`);  
        } else {  
            console.log("Deposit amount must be greater than 0.");  
        } // Method to withdraw money (controlled modification)  
    withdraw(amount) {  
        if (amount > 0 && amount <= this.#balance) {  
            this.#balance -= amount;  
            console.log(`$$ ${amount} withdrawn successfully.`);  
        } else {  
            console.log("Insufficient balance or invalid amount.");  
        } } // Private property  
    #balance;  
} // Using the BankAccount class  
const account = new BankAccount("Sangam", 1000);  
// Public access through methods  
console.log(account.getBalance()); // Output: The balance for Sangam is $1000.  
account.deposit(500); // Output: $500 deposited successfully.  
console.log(account.getBalance()); // Output: The balance for Sangam is $1500.  
account.withdraw(300); // Output: $300 withdrawn successfully.  
console.log(account.getBalance()); // Output: The balance for Sangam is $1200.  
// Trying to directly access the private property  
// console.log(account.#balance); // Error: Private field '#balance' must be declared in an enclosing class
```



# BROWSER OBJECT MODEL(BOM)

- The Browser Object Model (BOM) is used to interact with the browser.
- The default object of browser is window means you can call all the functions of window by specifying window or directly.

## Example:

```
<!DOCTYPE html>
<html>
<body>
<script>
//window.alert("Welcome to T3 skills cenete")
//is same as:
alert("Welcome to twksaa skills cenete")
</script>
</body>
</html>
```

## Output:

Welcome to T3 skills cenete

- You can use a lot of properties (other objects) defined underneath the window object like document, history, screen, navigator, location, innerHeight, innerWidth,

## ❖ Window Object:

- The window object represents a window in browser. An object of window is created automatically by the browser.
- Window is the object of browser, it is not the object of javascript. The javascript objects are string, array, date etc.
- The important methods of window object are as follows:

S.no	Method	Description
1	<b>alert()</b>	displays the alert box containing message with ok button.
2	<b>confirm()</b>	displays the confirm dialog box containing message with ok and cancel button.
3	<b>prompt()</b>	displays a dialog box to get input from the user.
4	<b>open()</b>	opens the new window
5	<b>close()</b>	closes the current window.
6	<b>setTimeout()</b>	performs action after specified time like calling function, evaluating expressions etc.

## 3. alert():

- **Description:** Displays an alert box with a specified message and an OK button. It is often used to inform the user of something.
- **Syntax:**  
alert(message);
  - **message:** The text string to display in the alert box.
- **Example-1:**  
alert("This is an alert box!");

**Example-2:**

```
<!DOCTYPE html>
<html>
<body>
<script type="text/javascript">
    function msg(){
        alert("Welcome!!!!");
    }
</script>
<input type="button" value="Click Here" onclick="msg()" />
</body>
</html>
```

**4. confirm():**

- **Description:** Displays a confirmation dialog box with a specified message, along with OK and Cancel buttons. It returns true if the user clicks OK, and false if the user clicks Cancel.

- **Syntax:**

```
let result = confirm(message);
message: The text string to display in the confirmation box.
```

- **Example-1:**

```
let userConfirmed = confirm("Do you want to proceed?");
if (userConfirmed) {
    // User clicked OK
} else {
    // User clicked Cancel
}
```

**Example-2:**

```
<!DOCTYPE html>
<html>
<body>
<script type="text/javascript">
    function msg(){
        let v= confirm("Are u sure?");
        if(v==true){
            alert("Your record deleted!");
        }
        else{
            alert("Cancel!!!!");
        }
    }
</script>
<input type="button" value="delete record" onclick="msg()" />
</body>
</html>
```



## 5. **prompt():**

- **Description:** Displays a dialog box that prompts the user to input a value. It returns the input value as a string. If the user clicks Cancel, it returns null.
- **Syntax:**  
let result = prompt(message, default);
  - **message:** The text string to display in the dialog box.
  - **default:** (Optional) The default input value.
- **Example-1:**

```
let userInput = prompt("Please enter your name:", "John Doe");
if (userInput !== null) {
    console.log("Hello, " + userInput);
}
```

### Example-2:

```
<!DOCTYPE html>
<html>
<body>
<script type="text/javascript">
function msg(){
    let v= prompt("Who are you?");
    alert("I am "+v);
}
</script>
<input type="button" value="Click Here" onclick="msg()"/>
</body>
</html>
```

## 6. **open():**

- **Description:** Opens a new browser window or tab.
- **Syntax** `window.open(URL, name, features);`

### Parameters

1. **URL:** The URL to open in the new window/tab. If omitted, a blank page is opened.
  2. **name:** The name of the new window. It can be used for targeting in links/forms.(optional)
    - **Example:** \_blank, \_self, \_parent, \_top, or a custom name.
  3. **features:** A string specifying the features of the new window, like size, position, and whether to include scrollbars, toolbars, etc.(optional)
- **Example-1:**

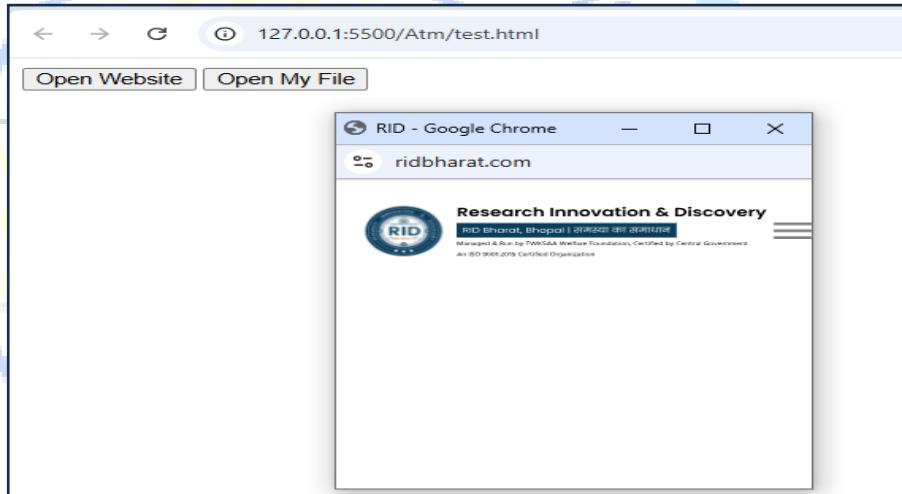
```
window.open("https://www.ridbharat.com", "_blank", "width=500,height=500")
```

### Example-2:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Window Open Example</title>
</head>

<body>
```

```
<!-- Button to open a new window -->
<button type="button" onclick="openWebsite()">Open Website</button>
<!-- Button to open your own file -->
<button type="button" onclick="openOwnFile()">Open My File</button>
<script>
    // Function to open a specific website (e.g., RID Bharat website) in a new window.
    function openWebsite() {
        window.open(
            "https://www.ridbharat.com", // URL to open
            "_blank", // Open in a new tab/window
            "width=300,height=300,left=100,top=50" // Specify dimensions and position
        );
    } // Function to open your own file (e.g., 'onlinetest.html') in a new window
    function openOwnFile() {
        window.open(
            "onlinetest.html", // Path to your file
            "OnlineTest", // Name of the new window
            "width=600,height=400,left=200,top=100" // Specify dimensions and position
        );
    }
</script> </body> </html>
```



### Example-3:

#### Index.html(main file)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Example</title>
</head>
<body>
    <button id="loginButton">Login (New Window)</button>
    <button id="loginButtonTab">Login (New Tab)</button>
    <script>
        document.getElementById("loginButton").addEventListener("click", function() {
            // Open a new window with the home.html file
            window.open("home.html", "newWindow", "width=600,height=400");
        });
        document.getElementById("loginButtonTab").addEventListener("click", function() {
            // Open a new tab with the home.html file
            window.open("home.html", "_blank");
        });
    </script> </body></html>
```

## 7. close():

- **Description:** The close() method is used to close the current window. It can only be called on windows that were opened by window.open() or similar methods.

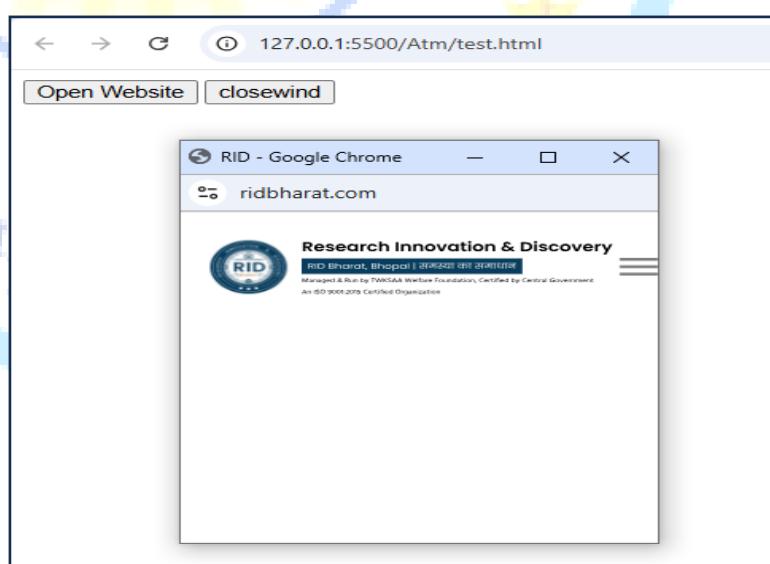
- **Syntax** .close();

- **Example-1:**

```
window.open('https://example.com', '_blank'); // Opens a new window  
window.close(); // Closes the current window
```

- **Example-2:**

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>Window Open Example</title>  
  </head>  
  <body>  
    <button type="button" onclick="openWebsite()">Open Website</button>  
    <!-- Button to close the file -->  
    <button type="button" onclick="closewebsite()">closewind</button>  
    <script>  
      let mywin  
      function openWebsite() {  
        mywin = window.open(  
          "https://www.ridbharat.com",  
          "_blank",  
          "width=300,height=300,left=100,top=150"  
        );  
      }  
      function closewebsite() {  
        mywin.close()  
      }  
    </script>  
  </body>  
</html>
```



## 8. setTimeout()

- **Description:** The setTimeout() method calls a function or evaluates an expression after a specified number of milliseconds. It's often used to execute code after a delay.

- **Syntax**

```
setTimeout(function, milliseconds);
```

- **function:** The function to execute after the timeout.

- **milliseconds:** The number of milliseconds to wait before executing the code.

**Example-1:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <button type="button" class="c1">display data</button>
    <button type="button" class="c2">stop</button>
    <h1 id="d1"></h1>
    <h1 id="d2"></h1>
<script>
const dpd=document.querySelector(".c1")
let setdata
dpd.addEventListener("click", ()=>{
    setdata=setTimeout(()=>{
        document.querySelector("#d1").innerHTML="After 3 second data will display"
    }, 3000)
})
const stp=document.querySelector(".c2")
stp.addEventListener("click", ()=>{
    clearTimeout(setdata)
    document.querySelector("#d2").innerHTML="your data will not display"
})
</script>
</body>
</html>
```

display data stop

After 3 second data will display

127.0.0.1:5500/bom/index.html

display data stop

your data will not display

```
<!DOCTYPE html>
<html>
<head>
<title>Window Close Example</title>
<script>
function openAndCloseWindow() {
    // Open a new window with the URL 'https://example.com'
    var newWindow = window.open('https://example.com', '_blank');

    // Close the new window after 5 seconds (5000 milliseconds)
    setTimeout(function() {
        if (newWindow) {
            newWindow.close(); // Closes the newly opened window
        } else {
            alert("The window could not be closed or was blocked by the browser.");
        }
    }, 5000);
}
</script>
</head>
<body>
<button onclick="openAndCloseWindow()">Open and Close Window</button>
</body>
</html>
```

## HISTORY OBJECT

- The JavaScript History object is a built-in object that represents the browser's session history. It provides methods and properties to navigate through the user's browsing history, such as going back and forward between visited pages and manipulating the browser's history stack.

- back():** This method moves the browser back one page in the session history. It is equivalent to clicking the "Back" button in the browser.

```
window.history.back();
```

- forward():** This method moves the browser forward one page in the session history. It is equivalent to clicking the "Forward" button in the browser.

- go([delta]):** The go() method allows you to navigate through the history by a specified number of steps. A positive delta value moves forward, and a negative value moves backward.

```
window.history.go(2); // Move forward two pages
```

```
window.history.go(-1); // Move backward one page
```

### ❖ Properties:

- length:** The length property represents the number of entries in the session history, which is the number of pages in the browser's history stack.

```
const historyLength = window.history.length;
```

- state:** This property allows you to associate a custom state object with a history entry. This state object can be accessed when the user navigates to that entry using the popstate event.

```
window.history.pushState({ data: custom state }, Custom Title, /new-page);
```

Here, { data: 'custom state' } is the custom state object associated with the history entry.

- pushState():** This method allows you to add a new state to the session history. This doesn't trigger a page load but instead changes the URL and adds an entry to the history stack.

```
window.history.pushState({ data: custom state }, Custom Title, /new-page);
```

- replaceState():** The replaceState() method is similar to pushState() but replaces the current state in the history stack with a new one, without adding a new entry.

### Example:

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript History Object Example</title>
</head>
<body>
    <h1>JavaScript History Object Example</h1>
    <p>Use buttons below to interact with browser's history.</p>
    <button onclick="goBack()">Go Back</button>
    <button onclick="goForward()">Go Forward</button>
    <button onclick="goSteps(2)">Go Forward 2 Steps</button>
    <button onclick="goSteps(-1)">Go Back 1 Step</button>
    <button onclick="addState()">Add State</button>
    <button onclick="replaceState()">Replace State</button>
    <div class="info">
        <p><strong>History Length:</strong> <span id="historyLength">0</span></p>
        <p><strong>Current State:</strong> <span id="currentState">None</span></p>
    </div>
```

```
<style>
    body {
        font-family: Arial, sans-serif;
    }
    button {
        margin: 5px;
        padding: 10px;
        font-size: 16px;
    }
    .info {
        margin-top: 20px;
        padding: 10px;
        border: 1px solid #ccc;
        background-color: #f9f9f9;
    }
</style>
```



```
<script> // Function to navigate back in history
function goBack() {
    window.history.back();
    updateInfo();
} // Function to navigate forward in history
function goForward() {
    window.history.forward();
    updateInfo();
} // Function to navigate by a specified number of steps
function goSteps(steps) {
    window.history.go(steps);
    updateInfo();
} // Function to add a new state to the history
function addState() {
    const newState = { data: 'Custom State Data' };
    const newTitle = 'Custom Page';
    const newUrl = `/custom-page-${Math.random().toString(36).substring(7)}`;
    window.history.pushState(newState, newTitle, newUrl);
    updateInfo();
} // Function to replace the current state
function replaceState() {
    const newState = { data: 'Replaced State Data' };
    const newTitle = 'Replaced Page';
    const newUrl = `/replaced-page-${Math.random().toString(36).substring(7)}`;
    window.history.replaceState(newState, newTitle, newUrl);
    updateInfo();
} // Function to update the displayed history information
function updateInfo() {
    const historyLength = window.history.length;
    const currentState = JSON.stringify(window.history.state) || 'None';
    document.getElementById('historyLength').textContent = historyLength;
    document.getElementById('currentState').textContent = currentState;
} // Initialize the info display on page load
updateInfo();
// Listen for popstate events to update info
window.addEventListener('popstate', updateInfo);
</script>
</body>
</html>
```

## JavaScript History Object Example

Use the buttons below to interact with the browser's history.

Go Back

Go Forward

Go Forward 2 Steps

Go Back 1 Step

Add State

Replace State

History Length: 2

Current State: {"data":"Replaced State Data"}



**RID BHARAT**

## **NAVIGATOR OBJECT**

- Navigator object is a built-in object in web browsers that provides information about the client's web browser and operating system. Developers can use the Navigator object to access various properties and methods to determine and control the user's browsing environment.
- **key properties and methods of the Navigator object:**
  1. **navigator.userAgent:** Returns a string representing the user agent header sent by the browser. This string often contains information about the browser's name and version.

```
Const userAgent = navigator.userAgent;
console.log(userAgent);
```
  2. **navigator.appName:** Returns the name of the browser.

```
Const browserName = navigator.appName;
console.log(browserName);
```
  3. **navigator.appVersion:** Returns the version of the browser.

```
Const browserVersion = navigator.appVersion;
console.log(browserVersion);
```
  4. **navigator.platform:** Returns the name of the client's operating system.

```
const platform = navigator.platform;
console.log(platform);
```
  5. **navigator.language:** Returns the preferred language of the user's browser.

```
Const preferredLanguage = navigator.language;
console.log(preferredLanguage);
```
  6. **navigator.cookieEnabled:** Returns a Boolean indicating whether cookies are enabled in the user's browser.

```
Const cookiesEnabled = navigator.cookieEnabled;
console.log(cookiesEnabled);
```
  7. **navigator.onLine:** Returns a Boolean indicating whether the browser is currently online (connected to the internet).

```
Const onlineStatus = navigator.onLine;
console.log(onlineStatus);
```

### ❖ **Methods:**

1. **navigator.geolocation** Allows access to the user's geographic position if the user grants permission. This can be used for location-based services.

```
if ('geolocation' in navigator) {
  navigator.geolocation.getCurrentPosition(function(position) {
    console.log('Latitude: ' + position.coords.latitude);
    console.log('Longitude: ' + position.coords.longitude);
  });
} else {
  console.log('Geolocation is not supported in this browser.');
}
```
2. **navigator.vibrate():** Allows the device to vibrate if supported.

```
if ('vibrate' in navigator) {
  navigator.vibrate(200); // Vibrate for 200 milliseconds
} else {
  console.log('Vibration is not supported in this browser.');}
```

3. **navigator.clipboard:** Provides access to the clipboard API, allowing you to read and write to the clipboard if the user grants permission.

```
if ('clipboard' in navigator) {  
    navigator.clipboard.writeText('Text to copy to clipboard').then(function() {  
        console.log('Text copied to clipboard');  
    }).catch(function(err) {  
        console.error('Failed to copy text: ', err);  
    });  
} else {  
    console.log('Clipboard API is not supported in this browser.');
```

**Example-1:**

```
<!DOCTYPE html>  
<html><head>  
<title>JavaScript Navigator Object Example</title>  
</head><body>  
<h1>Navigator Object Example</h1>  
<p><strong>User Agent:</strong><span id="userAgentInfo"></span></p>  
<p><strong>Browser Name:</strong><span id="browserNameInfo"></span></p>  
<p><strong>Browser Version:</strong><span id="browserVersionInfo"></span></p>  
<p><strong>Platform:</strong><span id="platformInfo"></span></p>  
<p><strong>Preferred Language:</strong><span id="languageInfo"></span></p>  
<p><strong>Cookies Enabled:</strong><span id="cookiesEnabledInfo"></span></p>  
<p><strong>Online Status:</strong><span id="onlineStatusInfo"></span></p>  
<script>  
    // Accessing properties of the Navigator object  
    document.getElementById('userAgentInfo').textContent = navigator.userAgent;  
    document.getElementById('browserNameInfo').textContent = navigator.appName;  
    document.getElementById('browserVersionInfo').textContent = navigator.appVersion;  
    document.getElementById('platformInfo').textContent = navigator.platform;  
    document.getElementById('languageInfo').textContent = navigator.language;  
    document.getElementById('cookiesEnabledInfo').textContent = navigator.cookieEnabled ?  
        'Enabled' : 'Disabled';  
    document.getElementById('onlineStatusInfo').textContent = navigator.onLine ? 'Online' :  
        'Offline';  
</script></body></html>
```

**Output:**

Navigator Object Example

User Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/116.0.0.0 Safari/537.36

Browser Name: Netscape

Browser Version: 5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/116.0.0.0 Safari/537.36

Platform: Win32

Preferred Language: en-US

Cookies Enabled: Enabled

Online Status: Online



# **JAVASCRIPT SCREEN OBJECT**

- The JavaScript Screen object is a built-in object that provides information about the user's screen or display. It contains properties that allow you to retrieve details such as screen dimensions, color depth, and whether the user's screen is a mobile device or a desktop monitor.

## ❖ **Properties:**

- screen.width:** Returns the width of the screen in pixels.  
Const sl = screen.width;
- screen.height:** Returns the height of the screen in pixels.  
Const sh = screen.height;
- screen.availWidth:** Returns the available width of the screen in pixels, excluding any system taskbars or docks.  
Const asw = screen.availWidth;
- screen.availHeight:** Returns the available height of the screen in pixels, excluding any system taskbars or docks.  
Const ash = screen.availHeight;
- screen.colorDepth:** Returns the color depth of the screen in bits per pixel. This indicates the number of colors a screen can display.  
Const colorDepth = screen.colorDepth;
- screen.pixelDepth:** Similar to `colorDepth`, this property also returns color depth of screen.  
Const pd = screen.pixelDepth;
- screen.orientation:** Returns an object that represents the orientation of the screen. It includes properties like type (e.g., "landscape-primary" or "portrait-secondary") and angle (the rotation angle in degrees).  
const orientation = screen.orientation;
- screen.devicePixelRatio:** Returns the ratio of the physical pixels to CSS pixels on the screen. It's often used for responsive design and detecting high-density displays (e.g., Retina displays).  
Const pixelRatio = screen.devicePixelRatio;
- screen.mobile:** A non-standard property that indicates whether the user's device is a mobile device. It's not supported in all browsers and is generally less reliable than other methods for detecting mobile devices.  
Const isMobile = screen.mobile;

## **Example:**

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Screen Object Example</title>
</head>
<body>
<h1>Screen Object Example</h1>
<p><strong>Screen Width:</strong><span id="screenWidth"></span> pixels</p>
<p><strong>Screen Height:</strong><span id="screenHeight"></span> pixels</p>
<p><strong>Available Width:</strong><span id="availWidth"></span> pixels</p>
<p><strong>Available Height:</strong><span id="availHeight"></span> pixels</p>
<p><strong>Color Depth:</strong><span id="colorDepth"></span> bits per pixel</p>
<p><strong>Pixel Depth:</strong><span id="pixelDepth"></span> bits per pixel</p>
```



```
<p><strong>Device Pixel Ratio:</strong><span id="pixelRatio"></span></p>
<p><strong>Orientation:</strong><span id="orientationType"></span></p>
<p><strong>Orientation Angle:</strong><span id="orientationAngle"></span> degrees</p>
<script>
    // Accessing properties of the Screen object
    document.getElementById('screenWidth').textContent = screen.width;
    document.getElementById('screenHeight').textContent = screen.height;
    document.getElementById('availWidth').textContent = screen.availWidth;
    document.getElementById('availHeight').textContent = screen.availHeight;
    document.getElementById('colorDepth').textContent = screen.colorDepth;
    document.getElementById('pixelDepth').textContent = screen.pixelDepth;
    document.getElementById('pixelRatio').textContent = screen.devicePixelRatio;

    // Accessing screen orientation properties
    const orientation = screen.orientation;
    document.getElementById('orientationType').textContent = orientation.type;
    document.getElementById('orientationAngle').textContent = orientation.angle;
</script>
</body>
</html>
```

**Output:**

Screen Object Example  
Screen Width: 1280 pixels  
Screen Height: 800 pixels  
Available Width: 1280 pixels  
Available Height: 752 pixels  
Color Depth: 24 bits per pixel  
Pixel Depth: 24 bits per pixel  
Device Pixel Ratio:  
Orientation: landscape-primary  
Orientation Angle: 0 degrees

# **COOKIES**

- Cookies in JavaScript are small pieces of data that websites can store on a user's device.
- They are often used for various purposes, such as tracking user sessions, storing user preferences, and maintaining stateful information between HTTP requests.
- A cookie is an amount of information that persists between a server-side and a client-side.

## **❖ How Cookies Works?**

- step-by-step explanation of how cookies work:
  - **Server-Side Creation:** When a user interacts with a website, the web server can create a cookie by including a Set-Cookie HTTP response header in its response to the user's browser. This header contains information about the cookie, including its name, value, and various optional attributes.
  - For example, a server can respond with a Set-Cookie header like this:  
Set-Cookie: username=Sangam; expires=Thu, 01 Jan 2025 00:00:00 UTC; path=/; domain=mywebsite.com; secure
  - **Storage in the Browser:** Once received, the browser stores this cookie data locally on the user's device. The data is typically stored in a text file or a similar data structure.
  - **Automatic Sending:** From this point on, whenever the user makes an HTTP request to the same domain (e.g., by clicking on links or loading pages), the browser automatically includes all relevant cookies for that domain in the Cookie HTTP request header.
- For example, if the user visits another page on the same website, the browser sends an HTTP request like this: GET /somepage HTTP/1.1
  - **Host:** mywebsite.com
  - **Cookie:**username=Sangam; othercookie=value
- **note:** The Cookie header contains all the cookies associated with that domain.
- **Server-Side Usage:** On the server, when it receives an HTTP request, it can access the Cookie header to retrieve the cookies sent by the user's browser. The server can then read and interpret the cookie data to determine, for example, the user's identity, preferences, or session information.
- **Updating and Deleting Cookies:** The server can also update or delete cookies by sending new Set-Cookie headers with the HTTP response. To delete a cookie, the server typically sets its expiration date to a date in the past.
- **Client-Side Access:** Cookies can also be accessed and manipulated on the client-side using JavaScript. This allows web developers to read and modify cookies as needed for client-side functionality, such as remembering user preferences or managing client-side sessions.

### **Example:**

```
<!DOCTYPE html>
<html>
<head></head> <body>
<input type="button" value="setCookie" onclick="setCookie()">
<input type="button" value="getCookie" onclick="getCookie()">
<script>
    function setCookie()
    { document.cookie="username=Sangam Kumar";  }
    function getCookie()
    {
```

```
if(document.cookie.length!=0)
{ alert(document.cookie); }
else
{ alert("Cookie not available");
} } </script> </body> </html>
```

**Example-2:**

```
<!DOCTYPE html>
<html>
<head> </head> <body>
<select id="color" onchange="display()">
<option value="Select Color">Select Color</option>
<option value="yellow">Yellow</option>
<option value="green">Green</option>
<option value="red">Red</option>
</select>
<script type="text/javascript">
    function display()
    {
        var value = document.getElementById("color").value;
        if (value != "Select Color")
        { document.bgColor = value;
            document.cookie = "color=" + value;
        } } n window.onload = function ()
        {
            if (document.cookie.length != 0)
            {
                var array = document.cookie.split("=");
                document.getElementById("color").value = array[1];
                document.bgColor = array[1];
            } } </script> </body> </html>
```

❖ **Cookies attributes:**

- Cookies in JavaScript can have various attributes that control their behavior.
- These attributes are specified when creating a cookie using the `document.cookie` property.

**1. Name and Value:**

- name=value: The name and value pair is the basic content of a cookie. It represents the data associated with the cookie.

**2. Expires:**

- expires=date: Specifies the expiration date and time for the cookie. After this date and time, cookie will be deleted. The date should be in the format "Day, DD Mon YYYY HH:MM:SS GMT."
- document.cookie = "username=John; expires=Thu, 01 Jan 2025 00:00:00 GMT";

**3. Max-Age:**

- max-age=seconds: Specifies the maximum age of the cookie in seconds. After the specified number of seconds, the cookie will expire.

document.cookie = "sessionToken=12345; max-age=3600"; // Expires in 1 hour

**4. Path:** path=path: Defines the path for which the cookie is valid. The cookie will only be sent to the server for URLs that match this path.

- document.cookie = "theme=dark; path=/myapp";



#### 5. Domain:

- domain=domain: Specifies the domain for which the cookie is valid. By default, cookies are associated with the domain of the current page.
- document.cookie = "auth=123; domain=mywebsite.com";

#### 6. Secure:

- secure: When present, the cookie will only be sent over secure (HTTPS) connections. It helps protect sensitive information. document.cookie = "secureCookie=secret; secure";

#### 7. HttpOnly:

- HttpOnly: When present, the cookie cannot be accessed by JavaScript. This attribute enhances security by preventing client-side scripts from reading the cookie.
- document.cookie = "sessionId=abc123; HttpOnly";

#### 8. SameSite:

- SameSite: Specifies how cookies should be sent in cross-origin requests. This attribute helps prevent certain types of CSRF attacks. Possible values are "Strict", "Lax", or "None".
- document.cookie = "csrfToken=xyz; SameSite=Strict";

➤ These attributes can be combined when setting a cookie to control its behavior. For example:

- document.cookie = "myCookie=value; expires=Thu, 01 Jan 2025 00:00:00 GMT; path=/; domain=mywebsite.com; secure; HttpOnly; SameSite=Strict";

#### ❖ Deleting a Cookie:

- You can delete a cookie in JavaScript by setting its expiration date to a past date.
- When a cookie's expiration date is in the past, the browser will automatically remove it.
- Different ways to delete a Cookie
- These are the following ways to delete a cookie:
- A cookie can be deleted by using expire attribute.
- A cookie can also be deleted by using max-age attribute.
- We can delete a cookie explicitly, by using a web browser.

**Example:** Create a cookie

```
document.cookie = "username=raj";
// Delete the cookie by setting its expiration date to the past
document.cookie = "username=; expires=Thu, 01 Jan 2010 00:00:00 GMT";
Example:
<!DOCTYPE html>
<html> <head> </head> <body>
<input type="button" value="Set Cookie" onclick="setCookie()">
<input type="button" value="Get Cookie" onclick="getCookie()">
<script>
function setCookie()
{ document.cookie="name=Martin Roy; expires=Sun, 20 Aug 2000 12:00:00 UTC"; }
function getCookie()
{ if(document.cookie.length!=0)
  { alert(document.cookie);
  }
  else
  { alert("Cookie not available");
  }
}
</script> </body> </html>
```

# JAVASCRIPT DEBUGGING

- JavaScript debugging is the process of identifying and fixing errors or issues in your JavaScript code. Debugging tools and techniques help you examine the code's behavior, trace the flow of execution, and pinpoint the source of problems.

- Here are common techniques and tools for debugging JavaScript:

## 1. Console Logging:

- Use `console.log()`, `console.error()`, and `console.warn()` to print values, variables, and messages to the browser's console. This is a quick and effective way to inspect the state of your code.
- `console.log("Debugging message:", someVariable);`

## 2. Breakpoints:

- Set breakpoints in your code using the browser's developer tools. Breakpoints pause execution at a specific line, allowing you to inspect variables and step through code.

## 3. Step Through Code:

- Use the debugging tools to step through code one line at a time. You can step into functions, step over function calls, and step out of functions to understand the flow of execution.

## 4. Inspect Variables:

- Examine the values of variables and objects in the scope. Most debugging tools provide a "Variables" or "Scope" panel where you can inspect and modify variables.

## 5. Conditional Breakpoints:

- Set breakpoints that only trigger when a specific condition is met. This can help you debug issues that occur under certain circumstances.

## 6. Call Stack:

- The call stack shows the sequence of function calls that led to the current point in your code. Understanding the call stack can help you identify where an error occurred.

## 7. Console Assertions:

- Use `'console.assert()'` to check if a condition is true and log an error message if it's not. This is helpful for verifying assumptions in your code.

- `console.assert(x > 0, "x should be greater than 0");`

## 8. Try-Catch Blocks:

- Wrap code in try-catch blocks to catch and handle exceptions gracefully. This prevents errors from crashing your entire application.

```
try {  
    // Code that may throw an error  
} catch (error) { // Handle the error  
}
```

## 9. Debugger Statement:

- Insert the `'debugger'` statement in your code to force a breakpoint at a specific location. This is useful for triggering debugging when needed.

```
function someFunction() {  
    // ...  
    debugger; // Pause execution here  
    // ...  
}
```

## 10. Error Messages and Stack Traces:

- Pay attention to error messages and stack traces in the console. They provide valuable information about what went wrong and where.

## **11. Browser Developer Tools:**

- Each web browser provides developer tools that include debugging features. Commonly used browsers like Chrome, Firefox, and Edge have robust developer toolsets for JavaScript debugging.

## **12. External Debugging Tools:**

- Consider using external JavaScript debugging tools like Visual Studio Code (with browser extensions), Firefox Developer Edition, or dedicated IDEs for JavaScript development.

### **Example:**

- examples of common JavaScript debugging scenarios along with techniques to address them using various debugging methods:

### **1. Syntax Error:**

```
// Example with a syntax error
const greeting = "Hello";
console.log(greeting; // Missing closing parenthesis
// Debugging: You'll see a syntax error in the browser's console.
- In this case, the browser's console will display a syntax error, and you can easily spot the issue in the code.
```

### **2. Variable Inspection:**

```
// Example with variable inspection
let x = 5;
let y = 10;
let z = x + y;
// Debugging: Place a breakpoint at the line with "let z = x + y;" and inspect the values of x, y, and z in the debugger's Variables panel.
- Set a breakpoint at the line of interest and inspect the variable values during runtime using the debugger.
```

### **3. Console Logging:**

```
// Example with console logging
function divide(a, b) {
  console.log(`Dividing ${a} by ${b}`);
  return a / b;
}
divide(10, 0); // Division by zero
// Debugging: Use console.log to print debug information. In this case, you'll see the log message before the error.
- Use `console.log()` to log messages and variable values to the console for debugging purposes.
```

### **4. Conditional Breakpoints:**

```
// Example with a conditional breakpoint
function findElement(arr, target) {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      console.log(`Found ${target} at index ${i}`);
      return i;
    }
  }
  return -1;
```



```
}
```

```
const numbers = [1, 3, 5, 7, 9];
```

```
findElement(numbers, 5); // Debugging: Set a breakpoint with a condition (e.g., i === 2) to stop execution when i is 2.
```

- Set a conditional breakpoint to pause code execution when specific conditions are met, allowing you to inspect the program state at that point.

### 5. Try-Catch for Error Handling:

```
// Example with try-catch for error handling
```

```
try {
```

```
    // Code that may throw an error
```

```
const result = someFunction();
```

```
console.log(`Result: ${result}`);
```

```
} catch (error) {
```

```
    console.error(`An error occurred: ${error.message}`);
```

```
}
```

```
// Debugging: Wrap code that might throw an error in a try-catch block to handle and log errors gracefully.
```

```
function someFunction() {
```

```
    return undefinedVariable; // This will throw a ReferenceError
```

```
}
```

```
- Use try-catch blocks to catch and handle errors, allowing your code to continue executing even if an error occurs.
```

### 6. Using the `debugger` Statement:

```
// Example using the debugger statement
```

```
function complexCalculation(a, b) {
```

```
    let result = a * b;
```

```
    debugger; // Set a breakpoint here
```

```
    result = result + 10;
```

```
    return result;
```

```
}
```

```
const result = complexCalculation(5, 3);
```

```
// Debugging: When the code reaches the "debugger" statement, execution will pause, and you can inspect variable values and the call stack.
```

- Insert the `debugger` statement in your code to force a breakpoint at a specific location for detailed debugging.



# JAVASCRIPT PROMISES

- JavaScript Promises are a way to handle asynchronous operations in a more structured and manageable manner.
- Promises in real-life express a trust between two or more persons and an assurance that a particular thing will surely happen.
- In JS, a Promise is an object which ensures to produce a single value in the future (when required).
- They provide a way to represent a value that may not be available yet but will be resolved at some point in the future, either successfully or with an error.
- Promises have become a fundamental part of JavaScript for dealing with asynchronous tasks like fetching data from a server, reading files, or handling user interactions.

## ❖ A Promise can have three states:

1. **Pending:** The initial state, indicating that the Promise is still waiting for the result.
2. **Fulfilled (Resolved):** The state when the asynchronous operation is successfully completed, and the Promise holds a resolved value.
3. **Rejected:** The state when an error occurs during the asynchronous operation, and the Promise holds a reason for the rejection.

## ❖ How to use Promises in JavaScript:

### ➤ Creating a Promise

- You can create a Promise using the Promise constructor, which takes a single argument, a function called the executor. The executor function has two parameters: resolve and reject. You call these functions to indicate whether the Promise should be fulfilled or rejected.

### Example:

```
const myPromise = new Promise((resolve, reject) => {
    // Asynchronous operation
    setTimeout(() => {
        const randomNumber = Math.random();
        if (randomNumber > 0.5) {
            resolve(randomNumber); // Resolve with a value
        } else {
            reject("Error: Number too small"); // Reject with an error message
        }
    }, 1000); // Simulating an asynchronous operation
});
```

Using .then() and .catch()

- Once you have a Promise, you can use the .then() method to specify what to do when the Promise is fulfilled, and you can use the .catch() method to specify what to do when the Promise is rejected.

### Example:

```
myPromise
    .then((result) => {
        console.log("Promise fulfilled with result:", result);
    })
    .catch((error) => {
        console.error("Promise rejected with error:", error);
    });

```



Example with Output

```
// Creating a Promise
const myPromise = new Promise((resolve, reject) => {
    // Simulating an asynchronous operation
    setTimeout(() => {
        const randomNumber = Math.random();
        if (randomNumber > 0.5) {
            resolve(randomNumber); // Resolve with a value
        } else {
            reject("Error: Number too small"); // Reject with an error message
        }
    }, 1000);
});
// Using .then() and .catch() to handle the Promise
myPromise
.then((result) => {
    console.log("Promise fulfilled with result:", result);
})
.catch((error) => {
    console.error("Promise rejected with error:", error);
});
```

#### **Output:**

```
Promise fulfilled with result: [random number]
If the random number is 0.5 or smaller, you'll see:
Promise rejected with error: Error: Number too small
```

**Note:** In this example, the Promise simulates an asynchronous operation with a random outcome.

Asynchronous in JavaScript:

- Asynchronous is a programming paradigm that allows you to execute code independently of the main program flow.
- It enables non-blocking operations, meaning that instead of waiting for an operation to complete, the program can continue executing other tasks. Asynchronous JavaScript is essential for handling tasks like network requests, file reading, timers, and user interactions without freezing the user interface.

➤ Here are key concepts and techniques for asynchronous programming in JavaScript

#### **1. Callbacks:**

- Callbacks are functions passed as arguments to other functions. They are executed when an asynchronous operation is complete.

#### **Example 1: Using Callbacks**

```
function fetchData(callback) {
    setTimeout(() => {
        callback("Data received");
    }, 1000);
}
fetchData((result) => {
    console.log(result); // Output: Data received
});
```

#### **2. Promises:**



- Promises are a more structured way to handle asynchronous operations and provide better error handling.

#### **Example 2: Using Promises**

```
function fetchData() {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const success = true;  
            if (success) {  
                resolve("Data received");  
            } else {  
                reject("Error: Data not received");  
            }  
        }, 1000);  
    });  
}  
fetchData()  
.then((result) => {  
    console.log(result); // Output: Data received  
})  
.catch((error) => {  
    console.error(error); // Output: Error: Data not received  
});
```

In this example, the `fetchData` function returns a Promise that resolves when the data is received and rejects if there's an error.

#### **3. Async/Await:**

- Async/await is a more readable way to work with Promises and make asynchronous code look synchronous.

#### **Example 3: Using Async/Await**

```
async function fetchData() {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const success = true;  
            if (success) {  
                resolve("Data received");  
            } else {  
                reject("Error: Data not received");  
            }  
        }, 1000);  
    });  
}  
async function main() {  
    try {  
        const result = await fetchData();  
        console.log(result); // Output: Data received  
    } catch (error) {  
        console.error(error); // Output: Error: Data not received  
    }  
}
```

```
}
```

```
main();
```

- In this example, the `main` function uses the `await` keyword to wait for the Promise to resolve or reject, making the code appear synchronous.

#### 4. Callbacks vs. Promises vs. Async/Await:

- Callbacks are less structured and can lead to callback hell or the pyramid of doom when dealing with multiple async operations. Promises and async/await provide cleaner and more maintainable code.

#### 5. Event Loop:

- The event loop is a central part of how asynchronous JavaScript works. It manages the execution of asynchronous tasks and callbacks.

#### 6. Timers:

- Timers like `setTimeout` and `setInterval` are commonly used for scheduling code to run asynchronously.

##### Example 4: Using `setTimeout`

```
console.log("Start");
setTimeout(() => {
  console.log("Inside setTimeout");
}, 1000);
console.log("End");
// Output:
// Start
// End
// Inside setTimeout (after 1 second)
```

**Note:** Asynchronous JavaScript is crucial for building responsive and efficient web applications, especially when dealing with I/O operations and interactions with external resources.

#### Complete Example:

##### ➤ Index.js:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Promise Example</title>
<link rel="stylesheet" href="Raj.css">
</head>
<body>
<div class="container">
<h1>Random User Data</h1>
<button id="fetchButton">Fetch Data</button>
<div id="userData">
<!-- User data will be displayed here -->
</div>
</div>
```

```
<script src="Raj.js"></script>
</body>
</html>
```

➤ **Raj.js**

```
// Function to fetch random user data using a Promise
function fetchData() {
    return new Promise((resolve, reject) => {
        fetch('https://jsonplaceholder.typicode.com/users/1')
            .then((response) => {
                if (!response.ok) {
                    throw new Error('Network response was not ok');
                }
                return response.json();
            })
            .then((data) => {
                resolve(data);
            })
            .catch((error) => {
                reject(error);
            });
    });
} // Function to display user data on the web page
function displayUserData(userData) {
    const userDataDiv = document.getElementById('userData');
    userDataDiv.innerHTML = `
        <h2>User Data</h2>
        <p><strong>Name:</strong> ${userData.name}</p>
        <p><strong>Email:</strong> ${userData.email}</p>
        <p><strong>Phone:</strong> ${userData.phone}</p>
        <p><strong>Website:</strong> ${userData.website}</p>
    `;
    // Add an event listener to the Fetch Data button
    const fetchButton = document.getElementById('fetchButton');
    fetchButton.addEventListener('click', () => {
        fetchData()
            .then((userData) => {
                displayUserData(userData);
            })
            .catch((error) => {
                const userDataDiv = document.getElementById('userData');
                userDataDiv.innerHTML = `<p>Error: ${error.message}</p>`;
            });
    });
}
```

➤ **Raj.css**

```
body {
    font-family: Arial, sans-serif;
    background-color: #f2f2f2;
    margin: 0;
```

```
padding: 0;
}
.container {
    background-color: #fff;
    max-width: 400px;
    margin: 0 auto;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 0 5px rgba(0, 0, 0, 0.2);
}
h1 {
    text-align: center;
}
button {
    background-color: #007bff;
    color: #fff;
    padding: 10px 20px;
    border: none;
    border-radius: 3px;
    cursor: pointer;
    display: block;
    margin: 0 auto;
}
button:hover {
    background-color: #0056b3;
}
#userData {
    margin-top: 20px;
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 3px;
    background-color: #f9f9f9;
}
```



# DOCUMENT OBJECT MODEL(DOM)

## ❖ What is the 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 dynamically.

## ❖ The Virtual DOM:

- React introduces the concept of a "virtual DOM." Instead of directly manipulating the actual DOM, React creates and maintains a lightweight virtual representation of it in memory. This virtual DOM is a tree-like structure that mirrors the actual DOM's structure.

## ❖ Tree-like Structure:

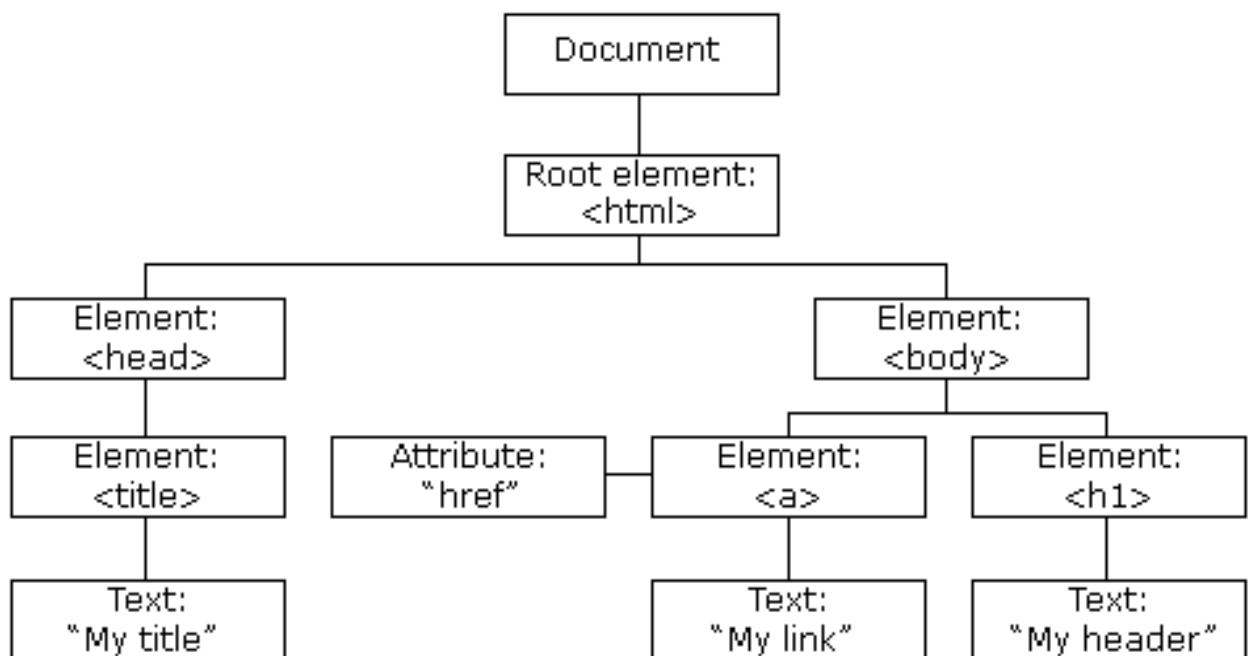
- The DOM represents an HTML or XML document as a hierarchical tree structure. Each element in the document, such as HTML tags, text, attributes, and comments, is represented as a node in this tree. These nodes are organized in a parent-child relationship, where the document itself is the root node.

## ❖ Node Types:

- There are several types of nodes in the DOM, including:

- Element Nodes:** Represent HTML elements like `<div>`, `<p>`, or `<a>`.
- Text Nodes:** Contain text within an element.
- Attribute Nodes:** Store attributes of elements.
- Comment Nodes:** Contain comments within the HTML.
- Document Node:** Represents the entire HTML document.

## HTML DOM Tree of Objects



### ❖ What is the HTML DOM?

- The HTML DOM is a standard object model and programming interface for HTML. It defines:
  - The HTML elements as objects
  - The properties of all HTML elements
  - The methods to access all HTML elements
  - The events for all HTML elements
- In other words: HTML DOM is a standard for how to get, change, add, or delete HTML elements.

#### Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<h1>POWER OF DOM</h1>
<p id="one"><b> With the object model, JavaScript gets all the power it needs to create dynamic HTML:</b></p>
<ul>
<li>JavaScript can change all the HTML elements in the page</li>
<li>JavaScript can change all the HTML attributes in the page</li>
<li>JavaScript can change all the CSS styles in the page</li>
<li>JavaScript can remove existing HTML elements and attributes</li>
<li>JavaScript can add new HTML elements and attributes</li>
<li>JavaScript can react to all existing HTML events in the page</li>
<li>JavaScript can create new HTML events in the page</li>
</ul>
<script>
    console.log(document)
    // With the HTML DOM, JavaScript can access and change all the elements of an HTML document
    console.log(document.getElementById("one"))
    console.log(document.getElementById("one").innerHTML)
    console.log(document.getElementById("one").innerText)
    document.getElementById("one").innerText="JS DOM features---"
</script>
</body>
</html>
```



# HTML DOM METHODS

- HTML DOM methods are actions. HTML DOM properties are values

**Note:** The HTML DOM can be accessed with JavaScript (and with other programming languages).

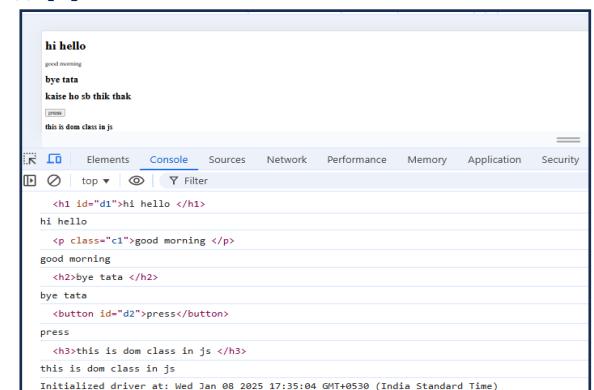
- The programming interface is the properties and methods of each object.
  - A **property is a value** that you can get or set (like changing the content of an HTML element).
  - A **method is an action** you can do (like add or deleting an HTML element).

## ❖ Methods for Selecting Elements:

- document.getElementById(id):** Selects an element by its id attribute.
- document.getElementsByClassName(className):** Returns a live HTMLCollection of elements with the specified class name.
- document.getElementsByTagName(tagName):** Returns a live HTMLCollection of elements with the specified tag name.
- document.querySelector(selector):** Selects first element that matches the CSS selector.
- document.querySelectorAll(selector):** Selects all elements that match the CSS selector.
- document.createElement(tagName):** Creates a new HTML element with specified tag name.
- document.createTextNode(text):** Creates a new text node with the specified text content.

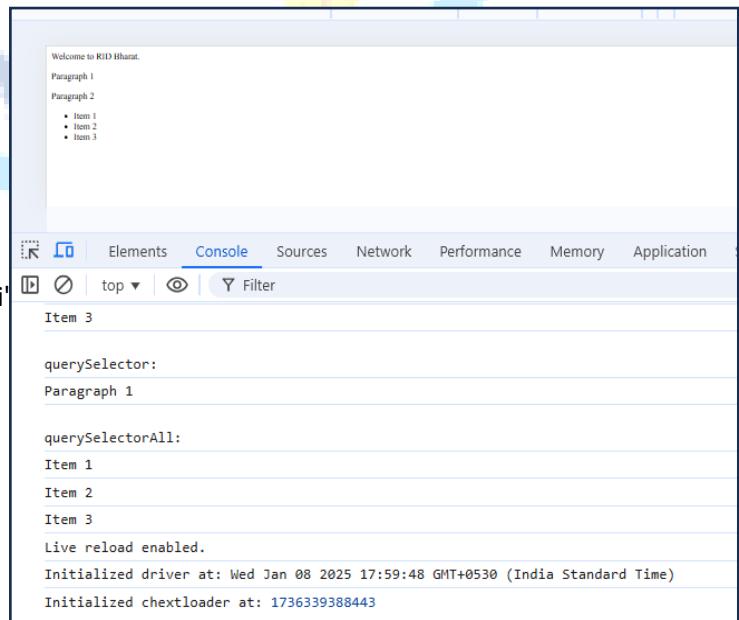
### Example-1:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1 id="d1">hi hello </h1>
    <p class="c1">good morning </p>
    <h2>bye tata </h2>
    <h2>kaise ho sb thik thak </h2>
    <button id="d2">press</button>
    <h3>this is dom class in js </h3>
    <p id="pp1">hi</p>
    <script> // DOM Method
        let a=document.getElementById("d1")
        console.log(a) console.log(a.textContent)
        let b=document.getElementsByClassName("c1")[0]
        console.log(b) console.log(b.textContent)
        let c=document.getElementsByTagName("h2")[0] // [0] this is index value
        console.log(c) console.log(c.textContent)
        let newd=document.querySelector("#d2")
        console.log(newd)
        console.log(newd.textContent)
        let f=document.querySelectorAll("h3")[0]
        console.log(f) console.log(f.textContent)
    </script>
</body></html>
```



**Example-2:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Selection Example</title>
</head>
<body>
    <div id="a">Welcome to RID Bharat.</div>
    <p class="c">Paragraph 1</p>
    <p class="c">Paragraph 2</p>
    <ul>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ul>
    <script>
        // getElementById
        const a = document.getElementById("a");
        console.log("getElementById:");
        console.log(a);
        console.log(a.textContent);
        // getElementsByClassName
        const b = document.getElementsByClassName("c");
        console.log("\ngetElementsByClassName:");
        for (const element of b) {
            console.log(element.textContent);
        }
        // getElementsByTagName
        const bc = document.getElementsByTagName("li");
        console.log("\ngetElementsByTagName:");
        for (const element of bc) {
            console.log(element.textContent);
        }
        // querySelector
        const cd = document.querySelector(".c");
        console.log("\nquerySelector:");
        console.log(cd.textContent);
        // querySelectorAll
        const ql = document.querySelectorAll("ul li");
        console.log("\nquerySelectorAll:");
        for (const element of ql) {
            console.log(element.textContent);
        }
    </script>
</body>
</html>
```



**Example-3:**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>DOM Manipulation Example</title>
</head>
<body>
<h1 id="hh1">Hello, Everyone</h1>
<p class="c">This is a paragraph with class "content".</p>
<p class="c">Another paragraph with class "content".</p>
<div class="c">This is a div with class "content".</div>
<ul>
<li>List item 1</li>
<li>List item 2</li>
<li>List item 3</li>
</ul>
<button onclick="changfun()">Click to Manipulate DOM</button>
<script>
function changfun() {
// 1. document.getElementById(id): Select the element by its ID
const udata = document.getElementById('hh1');
udata.style.color = 'blue';
udata.textContent = 'Hello, Every one Good Morning!';
// 2. document.getElementsByClassName(className): Select elements by their class name
const n = document.getElementsByClassName('c');
for (let i = 0; i < n.length; i++) {
n[i].style.backgroundColor = 'lightgreen';
n[i].textContent += ' (New updated data )';
}
// 3. document.getElementsByTagName(tagName): Select elements by their tag name
const m = document.getElementsByTagName('li');
for (let i = 0; i < m.length; i++) {
m[i].style.fontWeight = 'bold';
m[i].textContent += ' (this is new data )';
} } </script> </body> </html>
```

## Hello, Everyone

This is a paragraph with class "content".

Another paragraph with class "content".

This is a div with class "content".

- List item 1
- List item 2
- List item 3

[Click to Manipulate DOM](#)

## Hello, Every one Good Morning!

This is a paragraph with class "content". (New updated data )

Another paragraph with class "content". (New updated data )

This is a div with class "content". (New updated data )

- List item 1 (this is new data )
- List item 2 (this is new data )
- List item 3 (this is new data )

[Click to Manipulate DOM](#)



# Changing Element Content

There are following method are used for the changing the element

- 1) Set or get the plain text content of an `element.textContent,element.innerHTML,element.innerText`
- 2) Modifying Element Attributes with `setAttribute(name, value)`
- 3) Changing Element Styles with `style.property = value`
- 4) Creating and Appending New Elements in the DOM:  
`document.createElement(tagName)`
- 5) Removing Elements from the DOM Using `element.remove()`:
- 6) Replacing Elements in the DOM Using `element.replaceWith(newElement)`:

## 1. Set or get the plain text content of an element.

- `textContent`: it is used to set or get the plain text content of an element.
- `innerHTML`: Set or get HTML content of an element.it accepts content and tag behavior also
- `innerText`: Set or get HTML content of an element.it will not apply tag behavior

### ➤ **textContent:**

- Retrieves plain text, ignoring HTML tags.
- Updates the element to plain text without applying any HTML tag behavior.
- **Syntax:** `element.textContent; // Get the text content`  
`element.textContent = "New plain text"; // Set the text content`

- **Example:**

```
<div id="example">Hello <strong>RID Bharat</strong>!</div>
<script>
    let element = document.getElementById("example");
    console.log(element.textContent); Output: "Hello RID Bharat!"
    element.textContent = "New Text";
    console.log(element.textContent); Output: "New Text"
</script>
```

### ➤ **innerHTML:**

- Retrieves the full HTML structure, including tags.
- Updates the content and renders HTML tags properly **with the tag behaviour**.
- **Syntax:** `element.innerHTML; // Get the HTML content`  
`element.innerHTML = "<b>New HTML content</b>"; // Set the HTML content`

- **Example:**

```
<div id="example">Hello <span style="display:none;">Sonu</span> <strong>World</strong>!</div>
<script>
    let element = document.getElementById("example");
    console.log(element.innerHTML);
Output: " Hello <span style="display:none;">Sonu</span> <strong>World</strong>!"
    element.innerHTML = "<em>Updated HTML</em>";
    console.log(element.innerHTML); Output: "Updated HTML"
</script>
```

### ➤ **innerText:**

- Retrieves visible text only, ignoring text hidden via CSS (display: none).
- Updates visible text but doesn't apply or render HTML tag behavior.



- **Syntax:** element.innerText; **Get the visible text content**  
element.innerText = "New visible text"; **Set the visible text content**
- **Example:**

```
<div id="example">Hello <span style="display:none;">Sonu</span> World!</div>
<script>
  let element = document.getElementById("example");
  console.log(element.innerText); Output: "Hello World!"
  element.innerText = "Updated Text";
  console.log(element.innerText); Output: "Updated Text"
</script>
```

## 2. Modifying Element Attributes with setAttribute(name, value)

- The setAttribute method is used to change the value of an existing attribute or to add a new attribute to an HTML element. This method is versatile and works for any valid attribute.
- **Syntax:** element.setAttribute(attributeName, attributeValue);
  - ✓ **attributeName:** name of the attribute to modify or add (e.g., "class", "id", "src").
  - ✓ **attributeValue:** The value to assign to the attribute.
  - ✓ **outerHTML:** Returns the **entire HTML element**, including the element itself and its content (opening and closing tags, along with child elements or text).
- **Example:**

```
<!DOCTYPE html>
<html>
<head>
  <title>setAttribute Example</title>
</head>
<body>
  
  <script>
    // Select the image element
    let a = document.getElementById("image");
    console.log(a.outerHTML)
    console.log(a.innerHTML)
    // Change the 'src' attribute
    a.setAttribute("src", "example.jpg");
    // Add or modify the 'alt' attribute
    a.setAttribute("alt", "Updated Image Description");
    // Add a new 'title' attribute
    a.setAttribute("title", "Hover text for the image");
    console.log(a.outerHTML);
  </script> </body> </html>
```

**Output:** 

## 3. Changing Element Styles with style.property = value

- You can directly modify an element's inline styles by setting values for specific CSS properties using the style object in JavaScript. The property name in JavaScript should follow the camelCase convention (e.g., backgroundColor instead of background-color).
- **Syntax:** element.style.property = value;



- ✓ **property:** The CSS property you want to modify (written in camelCase).
- ✓ **value:** The value you want to assign to the property (as a string, including units like px if necessary).

**Example:**

```
<!DOCTYPE html>
<html>
<head>
<title>Change Element Style</title>
<style>
#example {
    width: 200px;
    height: 100px;
    background-color: lightblue;
    color: black;
    text-align: center;
    line-height: 100px;
}
</style>
</head>
<body>
<div id="example">Original Style</div>
<button onclick="change()">Change Style</button>
<script>
// Define the change function
function change() {
    // Select the element
    let element = document.getElementById("example");
    // Modify CSS styles using style.property
    element.style.backgroundColor = "yellow";
    element.style.color = "red";
    element.style.border = "2px solid green";
    element.style.fontSize = "20px";
    element.style.padding = "10px";
    // Output the modified element to the console
    console.log(element.outerHTML);
}
</script>
</body></html>
```

Original Style

Change Style

Original Style

Change Style

#### **4. Creating and Appending New Elements in the DOM:**

- **1. document.createElement(tagName)**
- **Description:** Creates a new HTML element with the specified tagName (e.g., div, p, button).
  - let newElement = document.createElement(tagName);
    - ✓ **tagName:** The name of the HTML tag to create as a string.
- **2. element.appendChild(newChild)**
  - Appends a new child element (newChild) to the specified parent element (element).
    - parentElement.appendChild(newChild);

**Example-1:**

```
<!DOCTYPE html>
<html>
<head> <title>Create and Append Elements</title>
</head>
<body>
  <div id="container">This is the container.</div>
  <button onclick="addElement()">Add New Element</button>
  <script>
    function addElement() {
      // Step 1: Create a new element
      let a = document.createElement("p");
      // Step 2: Add content to the new element
      a.textContent = "This is a dynamically added paragraph.";
      // Step 3: Select the parent element
      let bc = document.getElementById("container");
      // Step 4: Append the new element to the parent
      bc.appendChild(a);
      console.log(bc.outerHTML); // Log the updated container
    } </script> </body> </html>
```

**Example-2:**

```
<!DOCTYPE html>
<html>
<head>
  <title>Create and Append Dynamic Table</title>
</head>
<body>
  <div id="container">This is the container.</div>
  <button onclick="addTable()">Add Table</button>
  <script>
    function addTable() {
      // Step 1: Create a new table element
      let table = document.createElement("table");
      // Step 2: Create a table header row
      let headerRow = document.createElement("tr");
      let th1 = document.createElement("th");
      th1.textContent = "Header 1";
      let th2 = document.createElement("th");
      th2.textContent = "Header 2";
      headerRow.appendChild(th1);
      headerRow.appendChild(th2);
      // Step 3: Create a table data row
      let dataRow = document.createElement("tr");
      let td1 = document.createElement("td");
      td1.textContent = "Data 1";
      let td2 = document.createElement("td");
      td2.textContent = "Data 2";
```

```
<style>
  table {
    width: 50%;
    margin: 20px;
    border-collapse: collapse;
  }
  th, td {
    border: 1px solid black;
    padding: 10px;
    text-align: center;
  }
  th {
    background-color: #f0f0f0;
  }
  button {
    margin: 20px;
    padding: 10px 20px;
    background-color: #007bff;
    color: white;
    border: none;
    cursor: pointer;
    font-size: 16px;
  }
  button:hover {
    background-color: #0056b3;
  }
</style>
```

```

        dataRow.appendChild(td1);
        dataRow.appendChild(td2);
// Step 4: Append rows to the table
        table.appendChild(headerRow);
        table.appendChild(dataRow);
// Step 5: Append the table to the container
        let container = document.getElementById("container");
        container.appendChild(table);
        console.log(container.outerHTML); // Log the updated container
    } </script> </body></html>

```

This is the container.

Add Table

This is the container.

Header 1	Header 2
Data 1	Data 2

Add Table

## 5. Removing Elements from the DOM Using element.remove():

- The element.remove() method is used to remove an element from the DOM. Once removed, the element and its children are no longer part of the document.
- Syntax:** element.remove(); :- **element**: The DOM element you want to remove.
- Example:**

```

<!DOCTYPE html>
<html>
<head>
    <title>Remove Element Example</title>
</head>
<body>
    <div id="container">
        <p id="d1">This paragraph will be removed.</p>
    <button onclick="removeElement()">Remove Paragraph</button>
    </div>
    <script>
        function removeElement() {
            // Select the element to remove
            let paragraph = document.getElementById("d1");
            // Remove the element from the DOM
            paragraph.remove();
            console.log("Paragraph removed!");
        }
    </script>
</body>
</html>

```

```

<style>
#container {
    margin: 20px;
    padding: 10px;
    border: 2px solid black;
    background-color: #f9f9f9;
}
button {
    margin: 10px;
    padding: 10px 20px;
    background-color: #ff4d4d;
    color: white;
    border: none;
    cursor: pointer;
    font-size: 16px;
}
button:hover {
    background-color: #e60000;
}
</style>

```

This paragraph will be removed.

Remove Paragraph

Remove Paragraph



## 6. Replacing Elements in the DOM Using element.replaceWith(newElement):

- The `element.replaceWith(newElement)` method replaces an existing element (`element`) in the DOM with a new element (`newElement`). The original element is removed, and the new element takes its place.
- Syntax:** `element.replaceWith(newElement);`
  - ✓ `element`: The DOM element you want to replace.
  - ✓ `newElement`: The new DOM element that will replace the original one.
- Example:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Replace Element Example</title>
</head>
<body>
    <div id="container">
        <p id="oldElement">This is the old element.</p>
        <button onclick="replaceElement()">Replace Element</button>
    </div>
    <script>
        function replaceElement() {
            // Step 1: Select the element to be replaced
            let oldElement = document.getElementById("oldElement");
            // Step 2: Create the new element
            let newElement = document.createElement("p");
            newElement.id = "newElement"; // Set an ID for the new element
            newElement.textContent = "This is the new element.";
            // Step 3: Replace the old element with the new element
            oldElement.replaceWith(newElement);
            console.log("Element replaced!");
        }
    </script>
</body></html>
```

This is the old element.

Replace Element

```
<style>
    #container {
        margin: 20px;
        padding: 10px;
        border: 2px solid black;
        background-color: #f9f9f9;
    }
    #oldElement {
        color: red;
    }
    #newElement {
        color: green;
    }
    button {
        margin: 10px;
        padding: 10px 20px;
        background-color: #007bff;
        color: white;
        border: none;
        cursor: pointer;
        font-size: 16px;
    }
    button:hover {
        background-color: #0056b3;
    }
</style>
```

This is the new element.

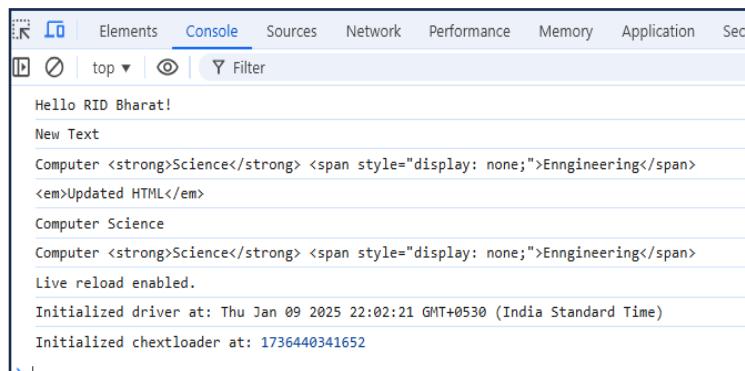
Replace Element



➤ Set or get the plain text content of an element:

➤ Example-1:

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body>
<div id="example">Hello <strong>RID Bharat</strong>!</div>
<h1></h1> <hr>
<div id="example-1">Computer <strong>Science</strong> <span style="display: none;">Enngineering</span> </div>
<h2></h2> <hr>
<div id="example-2">Computer <strong>Science</strong> <span style="display: none;">Enngineering</span> </div>
<h3></h3>
<script>
// textContent
let element = document.getElementById("example");
console.log(element.textContent); // Output: "Hello RID Bharat!"
element.textContent = "New Text ";
console.log(element.textContent); // Output: "New Text"
document.getElementsByTagName("h1")[0].textContent = "<i>Hello Every one</i>"
// Output: <i>Hello Every one</i>
// innerHTML
let ele = document.getElementById("example-1");
console.log(ele.innerHTML); // Output: Computer <strong>Science</strong> <span style="display: none;">Enngineering</span>
ele.innerHTML = "<em>Updated HTML</em>";
console.log(ele.innerHTML);
document.getElementsByTagName("h2")[0].innerHTML = "<i>Hello Every one</i>"
// innerText
let e = document.getElementById("example-2");
console.log(e.innerText) // output
console.log(e.innerHTML);
document.getElementsByTagName("h3")[0].innerHTML = "<i>Hello Every Student</i>"
</script>
</body> </html>
```



New Text

**<i>Hello Every one</i>**

Updated HTML

**Hello Every one**

Computer Science

**Hello Every Student**

### Example-2: Dom\_setatributes.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
<style>
.a {
    border: 5px solid red;
}
#two {
    background-color: yellow;
    color: blue;
    font-size: 20px;
}
</style>
</head>
<body>
<h1>Set the attributes </h1>
<p id="one">RID BHARAT </p>
<button onclick="myFun()">Click-1</button>
<button onclick="myFun1()">Click-2</button>
<script>
    function myFun() {
        let element =
document.getElementById("one")
        element.setAttribute("class", "a")
    }
    function myFun1() {
        let element =
document.getElementById("one")
        element.setAttribute("id", "two")
    }
</script>
</body> </html>
```

### Example-3:

```
> dom_write.html:
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
</head>
<style>
</style>
<body>
<h1 id="one"></h1>
<h4 id="two">
<button onclick="myFun()">Click to display</button>
</h4>
<script>
    let a=prompt("Enter the number ")
document.getElementById("one").innerText="Multiplication table of "+a;
    let n=a;
    function myFun(){
for(let i=1;i<10;i++){
document.write(n, " x ",i, " = ",n*i,"<br>")
    }
</script> </body> </html>
```

## Multiplication table of 10

[Click to display](#)

**10 x 1 = 10**  
**10 x 2 = 20**  
**10 x 3 = 30**  
**10 x 4 = 40**  
**10 x 5 = 50**  
**10 x 6 = 60**  
**10 x 7 = 70**  
**10 x 8 = 80**  
**10 x 9 = 90**  
**10 x 10 = 100**

## Set the attributes

RID BHARAT

[Click-1](#) [Click-2](#)

## Set the attributes

**RID BHARAT**

[Click-1](#) [Click-2](#)



#### Example-4: REMOVE Element

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<style>
p {
    height: 100px;
    width: 100px;
    border: 5px solid red;
}
</style>
<body>
<h1>document.removeChild(element)</h1>
<p>
</p>
<button onclick="myFun()">Remove div</button>
<script>
let a =
document.getElementsByTagName("p")[0]
function myFun() {
    if (a) {
        a.parentNode.removeChild(a)
    }
}
</script>
</body>
</html>
```

#### document.removeChild(element)



Remove div

#### Example-5: dom-replacechild.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<style>
#one {
    height: 100px;
    width: auto;
    border: 5px solid red;
}
#two {
    height: 100px;
    width: auto;
    border: 5px solid blue;
}
</style>
<body>
<h1>document.replaceChild(new, old)</h1>
<p id="one">This is old para </p>
<button onclick="myFun()">Click to replace</button>
<script>
function myFun() {
    let a = document.getElementById("one")
    let b = document.createElement("p")
    b.id = "two"
    let txt = document.createTextNode("This is new para")
    b.appendChild(txt)
    a.parentNode.replaceChild(b, a)
}
</script> </body> </html>
```

#### document.replaceChild(new, old)

This is old para

Click to replace

**Example-6: Dom.addevent.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
<h1>Adding Events Handlers</h1>
<p>Syntax:<br>
document.getElementById(id).onclick = function(){code}
</p>
<button id="one">Click</button>
<script>
// 
document.getElementById("one").onclick = function(){
    // alert("Button clicked successfully")
    //
let btn = document.getElementById("one")
btn.onclick = function(){
    alert("Click event added")
}
</script>
</body>
</html>
```

**Example-7: create.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<style>
img {
    height: 200px;
    width: 200px;
}
table {
    width: 100%;
    text-align: center;
}
</style>
<body>
<table border="1">
<thead>
<th>Name</th>
<th>Age</th>
<th>Image</th>
</thead>
<tbody id="t-body">
</tbody>
</table>
<script>
let data = [
    { "name": "Raj", age: 20, image: "https://i.postimg.cc/qMdDpVSH/one.jpg" },
    { "name": "Aryan", age: 30, image: "https://i.postimg.cc/j28D3rJn/two.jpg" },
    { "name": "Kazi", age: 40, image: "https://i.postimg.cc/Pq2nwrGD/three.jpg" },
    { "name": "Peter", age: 31, image: "https://i.postimg.cc/L6t9sSst/four.jpg" }
]
let t = document.getElementById("t-body")
for (let i = 0; i < data.length; i++) {
    let row = t.insertRow(i)
    let cell1 = row.insertCell(0)
    let cell2 = row.insertCell(1)
    let cell3 = row.insertCell(2)
    let pic = document.createElement("img")
    pic.src = data[i].image
    cell1.innerText = data[i].name
    cell2.innerText = data[i].age
    cell3.appendChild(pic)
}
</script> </body> </html>
```

**Example-08:**

➤ **multiplication table.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Tables</title>
</head>
<style>
#two {
    height: auto;
    width: auto;
    margin: 0 auto;
    border: 2px solid gold;
    background-color: pink;
}
</style>
<body>
    <h1 id="one"></h1>
    <button onclick="myFun()">Click to display</button>
    <script>
        let a = prompt("Enter the number ")
        document.getElementById("one").innerText = "Multiplication table of " + a;
        let num = a;
        function myFun() {
            let n = document.createElement("div")
            n.id = "two"
            document.body.appendChild(n)
            let op = ""
            for (let i = 1; i <= 10; i++) {
                op = op + num + " x " + i + " = " + num * i + "<br>";
            }
            console.log(op)
            document.getElementById("two").innerHTML = op
        }
    </script>
</body>
</html>
```

## Multiplication table of 10

Click to display

10 x 1 = 10  
10 x 2 = 20  
10 x 3 = 30  
10 x 4 = 40  
10 x 5 = 50  
10 x 6 = 60  
10 x 7 = 70  
10 x 8 = 80  
10 x 9 = 90  
10 x 10 = 100

**Example-09:**

➤ **Add\_class .html**

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>classList</title>
</head>
<style>
.one{
    border:5px solid red;}
.two{
    height: 100px;
    background-color: aqua;}
</style>
<body>
<h1>Adding multiple classes to an element</h1>
<p>className will be used to add single class to an element</p>
<p>classList handles multiple class names at one time</p>
<script>
    let p = document.createElement("p")
p.innerText="Dummy para text"
    // p.className="one"
p.classList.add("one")
p.classList.add("two")
document.body.appendChild(p)
</script></body> </html>
```

**Example-10: Important**

➤ **Todo\_index.html**

```
<!DOCTYPE html>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>To-DO List</title>
</head>
<style>
olli {
    font-size: 30px;
}
</style>
<body>
<h1>ToDo List</h1>
<div id="container">
<input type="text" id="inp">
<button id="add">Add</button>
<ol id="todos"></ol></div>
```

## Adding multiple classes to an element

className will be used to add single class to an element

classList handles multiple class names at one time

Dummy para text

```
<script>
let inpF = document.getElementById("inp")
let btn = document.getElementById("add")
let todo = document.getElementById("todos")
btn.addEventListener("click", () => {
    let res = document.createElement("li")
    res.innerText = inpF.value
    todo.appendChild(res)
    inpF.value = "" //clear the input field
    res.addEventListener("click", () => {
        res.style.textDecoration = "line-through"
    })
    res.addEventListener("dblclick", () => {
        todo.removeChild(res)
    })
})
</script> </body> </html>
```

## To Do List

 Add

1. HTML
2. CSS
3. JavaScript

### Example-11

#### Todo\_index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Todo Apps </title>
<style>
li button:hover {
    cursor: pointer;
    background-color: aqua;
}
</style>
</head>
<body>
<div>
<input type="text" id="inp" placeholder="Enter your task">
<button type="button" id="add">Add</button>
<ol type="1" id="todos"></ol>
</div>
<script>
let inpf = document.getElementById("inp");
let btn = document.getElementById("add");
let todo = document.getElementById("todos");
let v = null;
// Function to handle adding or editing todos
function addTodo() { // Edit functionality
    if (v) {
        v.innerHTML = inpf.value.trim();
        inpf.value = "";
        v = null;
        btn.innerHTML = "Add";
        return; }
```



**// Add functionality**

```

if (inpf.value.trim() !== "") {
    let res = document.createElement("li");
    let sp = document.createElement("span");
    sp.innerHTML = inpf.value;
    res.style.lineHeight = "2em";
    res.append(sp); // Create buttons
    let btn1 = document.createElement("button");
    let btn2 = document.createElement("button");
    let btn3 = document.createElement("button");
    let btn4 = document.createElement("button"); // Button properties
    btn1.innerHTML = "Mark";
    btn1.style.marginLeft = "10px";
    btn2.innerHTML = "Update";
    btn2.style.marginLeft = "10px";
    btn3.innerHTML = "Remove";
    btn3.style.marginLeft = "10px";
    btn4.innerHTML = "Complete";
    btn4.style.marginLeft = "10px"; // Append buttons to the list item
    res.append(btn1, btn2, btn3, btn4);
    todo.appendChild(res); // Clear input field
    inpf.value = "" // Button functionalities
    btn1.addEventListener("click", () => {
        res.style.color = "green";
        res.style.textDecoration = "underline";
    });
    btn2.addEventListener("click", () => {
        inpf.value = sp.innerHTML;
        v = sp;
        btn.innerHTML = "Edit";
    });
    btn3.addEventListener("click", () => {
        todo.removeChild(res);
    });
    btn4.addEventListener("click", () => {
        alert("Your data is updated successfully!");
    });
} else {
    alert("Input cannot be empty!");
}
}

```

```

// Event listener for the Add button
btn.addEventListener("click", addTodo);
// Event listener for the Enter key
inpf.addEventListener("keydown", (event) => {
    if (event.key === "Enter") {
        addTodo();
    }
});
</script>
</body>
</html>

```



1. HTML

2. CSS

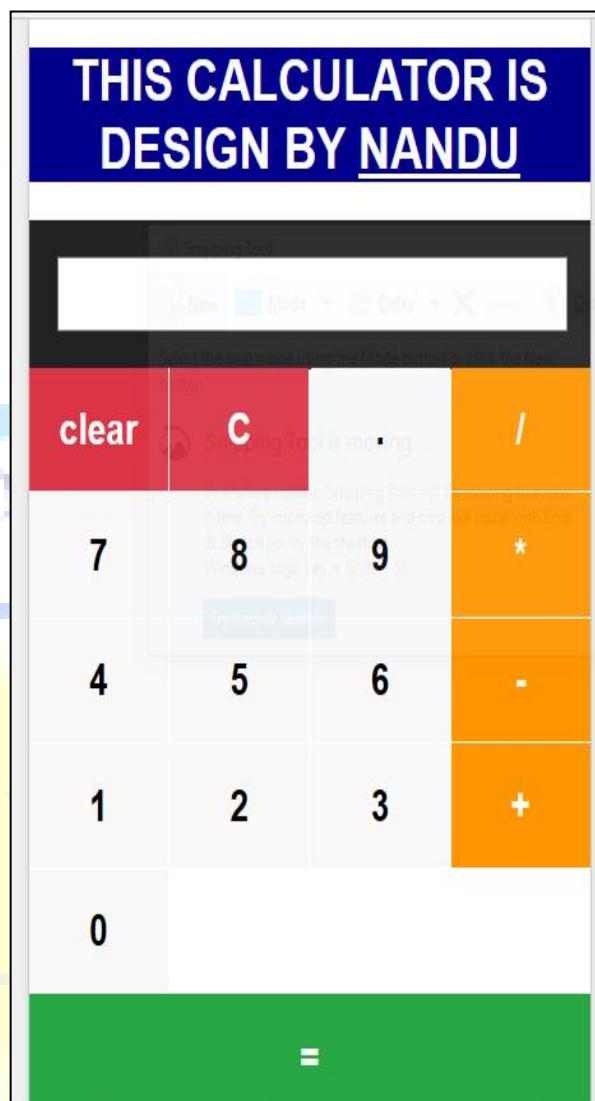


## Calculator project

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" />
<title>Calculator</title>
<style>
body {
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    margin: 0;
    background-color: #f0f0f0;
    font-family: Arial, sans-serif;
}
h1 {text-align: center;}
.calculator {
    border: 1px solid #ccc;
    border-radius: 10px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    overflow: hidden;
    width: 400px;
    background-color: white;
}
.display {
    background-color: #222;
    color: white;
    text-align: center;
    padding: 20px;
    font-size: 2em;
}
.display input {
    width: 100%;
    font-size: 30px;
    font-weight: bold;
}
.buttons {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    gap: 1px;
}
.buttons button {
    padding: 20px;
    font-size: 1.5em;
    font-weight: bold;
    border: none;
    background-color: #f9f9f9;
    cursor: pointer;
    transition: background-color 0.3s;
}
.buttons button:hover {
    background-color: #e0e0e0;
}
background-color: #e0e0e0;
.buttons button:active {
    background-color: #d0d0d0;
}
.buttons .operator {
    background-color: #ff9500;
    color: white;
}
.buttons .operator:hover {
    background-color: #ff8500;
}
.buttons .operator:active {
    background-color: #ff7500;
}
.buttons .equals {
    background-color: #28a745;
    color: white;
    grid-column: span 4;
}
.buttons .equals:hover {
    background-color: #218838;
}
.buttons .equals:active {
    background-color: #1e7e34;
}
.buttons .clear {
    background-color: #dc3545;
    color: white;
}
.buttons .clear:hover {
    background-color: #c82333;
}
.buttons .clear:active {
    background-color: #bd2130;
}
</style></head><body>
<div class="calculator">
<h1 style="background-color: darkblue; color: white;">THIS CALCULATOR IS DESIGN BY <strong><U> NANDU </U></strong></h1>
<div class="display"><input type="text" readonly /></div>
<div class="buttons">
    <button class="clear" onclick="clr()">clear</button>
    <button class="clear" onclick="backspace()">C</button>
    <button onclick="fun(this)">.</button>
    <button class="operator" onclick="fun(this)">/</button>
    <button onclick="fun(this)">7</button>
    <button onclick="fun(this)">8</button>
    <button onclick="fun(this)">9</button>
    <button class="operator" onclick="fun(this)">*</button>
    <button onclick="fun(this)">4</button>
    <button onclick="fun(this)">5</button>
    <button onclick="fun(this)">6</button>
    <button class="operator" onclick="fun(this)">-</button>
    <button onclick="fun(this)">1</button>
    <button onclick="fun(this)">2</button>
    <button onclick="fun(this)">3</button>
    <button class="operator" onclick="fun(this)">+</button>
    <button onclick="fun(this)">0</button>
</div>

```

```
<button class="equals"
    onclick="res()">>=</button>
  </div>
</div>
<script>
  let inp =
    document.getElementsByTagName("input")[0];
  function fun(e) {
    inp.value += e.innerText;
  }
  function res() {
    try {
      inp.value = eval(inp.value);
    } catch (e) {
      inp.value = "Error";
    }
  }
  function clr() {
    inp.value = "";
  }
  function backspace() {
    inp.value = inp.value.slice(0, -1);
  }
</script>
</body>
</html>
```



BHARAT

## ❖ Adding and Deleting Elements:

Method	Description
1. document.createElement(element)	Create an HTML element
2. document.removeChild(element)	Remove an HTML element
3. document.appendChild(element)	Add an HTML element
4. document.replaceChild(new, old)	Replace an HTML element
5. document.write(text)	Write into the HTML output stream

### Example:

```
<!DOCTYPE html>
<html>
<head>
<title>DOM Manipulation Example</title>
</head>
<body>
<h1>Shopping List</h1>
<ul id="shoppingList">
<li>Apples</li>
<li>Bananas</li>
<li>Oranges</li>
</ul>
<button onclick="addItem()">Add Item</button>
<button onclick="deleteItem()">Delete Item</button>
<script>
// Function to add a new item to the list
function addItem() {
    var shoppingList = document.getElementById("shoppingList");
    var newItem = document.createElement("li");
    newItem.textContent = "Grapes"; // Text for the new item
    shoppingList.appendChild(newItem);
}
// Function to delete an item from the list
function deleteItem() {
    var shoppingList = document.getElementById("shoppingList");
    var items = shoppingList.getElementsByTagName("li");
    if (items.length > 0) {
        shoppingList.removeChild(items[0]); // Delete the first item
    }
}
</script>
</body>
</html>
```

❖ **Adding Events Handlers:**

- Method Description
- `document.getElementById(id).onclick = function(){code}` Adding event handler code to an onclick event

**Syntax:** `document.getElementById(id).onclick = function() { code }`

1. **document.getElementById(id):** This method selects an HTML element by its unique id attribute.
2. **.onclick:** This property allows you to assign a function to handle the click event of the selected element.
3. **= function() { code }:** This is the function you define to execute when the click event occurs. The code inside the function specifies what should happen.

**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Basic Click Event</title>
</head>
<body>
    <button id="myButton">Click Me</button>
    <script>
        // JavaScript code goes here
    </script>
</body>
</html>
```

// Select the button element by its ID  
`document.getElementById('myButton').onclick = function() {  
 // Define what happens when the button is clicked  
 alert('Button was clicked!');  
};`

**Explanation:**

- `document.getElementById('myButton')` selects the button element with the ID myButton.
- `.onclick` assigns a function to handle the click event.
- The function displays an alert with the message "Button was clicked!" when the button is clicked.

## Finding HTML Objects

- Finding HTML objects is a fundamental concept in web development that involves locating and interacting with specific elements within an HTML document using JavaScript.
- This concept is essential for creating dynamic and interactive web pages.
- ❖ **Document Object Model (DOM):** The DOM is a programming interface for web documents. It represents the structure of an HTML document as a tree of objects, where each object corresponds to an element in the HTML. JavaScript can be used to manipulate this tree, which allows you to find and interact with HTML elements.
- ❖ **Selecting Elements:** To find HTML objects within the DOM, you typically start by selecting or referencing them. There are various methods and techniques to select elements, including:
  - ❖ **getElementById:** This method allows you to select an element by its unique id attribute.
  - ❖ **getElementsByClassName:** You can select elements based on their class names. Multiple elements with the same class can be selected.
  - ❖ **getElementsByTagName:** This method selects elements by their HTML tag name (e.g., "div," "p," "a").
  - ❖ **querySelector and querySelectorAll:** These methods use CSS-style selectors to select elements. querySelector returns the first matching element, while querySelectorAll returns a collection of all matching elements.
- ❖ **Traversing the DOM:** You can navigate the DOM tree by moving from one element to another using properties like parentNode, childNodes, nextSibling, and previousSibling. This is useful for finding elements relative to a known reference point.
- ❖ **Using Properties and Methods:** Once you have a reference to an HTML element, you can use various properties and methods to interact with it. Common operations include changing the content, modifying attributes, altering the styling, attaching event listeners, and more.
- ❖ **Event Handling:** After finding HTML objects, you can attach event handlers to them. Event handlers allow you to respond to user interactions, such as clicks, mouse movements, and keyboard inputs, making your web page interactive.

### Example:

```
<!DOCTYPE html>
<html>
<head>
<title>Finding HTML Objects Example</title>
</head>
<body>
<div id="myElementId">This is a div element with an id.</div>
<script>
    // Find the element with id "myElementId"
    var element = document.getElementById("myElementId");
    // Modify the element's content
    element.textContent = "Updated content";
</script>
</body>
</html>
```

**Note:** The first HTML DOM Level 1 (1998), defined 11 HTML objects, object collections, and properties.

These are still valid in HTML5

Property	Description	DOM
• document.anchors	Returns all <a> elements that have a name attribute	1
• document.applets	Deprecated	1
• document.baseURI	Returns the absolute base URI of the document	3
• document.body	Returns the <body> element	1
• document.cookie	Returns the document's cookie	1
• document.doctype	Returns the document's doctype	3
• document.documentElement	Returns the <html> element	3
• document.documentElementMode	Returns the mode used by the browser	3
• document.documentElementURI	Returns the URI of the document	3
• document.domain	Returns the domain name of the document server	1
• document.domConfig	Obsolete.	3
• document.embeds	Returns all <embed> elements	3
• document.forms	Returns all <form> elements	1
• document.head	Returns the <head> element	3
• document.images	Returns all <img> elements	1
• document.implementation	Returns the DOM implementation	3
• document.inputEncoding	Returns the document's encoding (character set)	3
• document.lastModified	Returns the date and time the document was updated	3
• document.links	Returns all <area> and <a> elements that have a href attribute	
• document.readyState	Returns the (loading) status of the document	3
• document.referrer	Returns the URI of the referrer (the linking document)	1
• document.scripts	Returns all <script> elements	3
• document.strictErrorChecking	Returns if error checking is enforced	3
• document.title	Returns the <title> element	1
• document.URL	Returns the complete URL of the document	1

### Example:

```
<!DOCTYPE html>
<html>
<head>
<title>DOM Properties and Methods Example</title>
</head>
<body>
<a name="section1">Section 1</a>
<a name="section2">Section 2</a>
<script>
    // document.anchors - Returns all <a> elements that have a name attribute
    var anchors = document.anchors;
    console.log("Anchors with name attribute:");
    for (var i = 0; i<anchors.length; i++) {
        console.log(anchors[i]);
    }
    // document.baseURI - Returns the absolute base URI of the document
    var baseURL = document.baseURI;
    console.log("Base URI of the document:", baseURL);

```

```
// document.body - Returns the <body> element
var bodyElement = document.body;
console.log("Body element:", bodyElement);
// document.cookie - Returns the document's cookie
var cookies = document.cookie;
console.log("Document's cookie:", cookies);
// document.doctype - Returns the document's doctype
var doctype = document.doctype;
console.log("Document's doctype:");
console.log(doctype);
// document.documentElement - Returns the <html> element
var htmlElement = document.documentElement;
console.log("HTML element:", htmlElement);
// document.embeds - Returns all <embed> elements
var embeds = document.embeds;
console.log("Embed elements:");
for (var i = 0; i<embeds.length; i++) {
    console.log(embeds[i]);
}
// document.head - Returns the <head> element
var headElement = document.head;
console.log("Head element:", headElement);
// document.images - Returns all <img> elements
var images = document.images;
console.log("Image elements:");
for (var i = 0; i<images.length; i++) {
    console.log(images[i]);
}
// document.title - Returns the <title> element
var titleElement = document.title;
console.log("Title element:", titleElement);
// document.URL - Returns the URL of the document
var documentURL = document.URL;
console.log("Document URL:", documentURL);
</script>
</body>
</html>
```

# JavaScript HTML DOM Elements

- JavaScript HTML DOM (Document Object Model) elements are the representation of HTML elements within a web page, accessible and manipulable through JavaScript.

## 1. Accessing DOM Elements:

- **getElementById:** This method allows you to select an element by its unique id attribute.
- **getElementsByClassName:** Select elements based on their class names.
- **getElementsByTagName:** Select elements by their HTML tag name (e.g., "div," "p," "a").
- **querySelector and querySelectorAll:** Use CSS-style selectors to select elements.
- **Traversing the DOM:** Navigate the DOM tree using properties like parentNode, childNodes, nextSibling, and previousSibling to find elements relative to a known reference point.

## 2. Properties and Methods:

- **textContent:** Gets or sets the text content of an element.
- **innerHTML:** Gets or sets the HTML content of an element.
- **attributes:** Access and manipulate attributes of an element.
- **style:** Access and manipulate CSS styles of an element.
- **addEventListener:** Attach event listeners to respond to user interactions.
- **removeEventListener:** Remove previously attached event listeners.
- **appendChild and removeChild:** Add and remove child elements within parent elements.
- **classList:** Access and manipulate the class attribute to add or remove CSS classes.
- **getAttribute and setAttribute:** Get and set attributes of elements.
- **value:** Access or change the value of form elements like input fields and textareas.
- **parentNode and childNodes:** Access a node's parent and child elements.

## 3. Common DOM Element Properties:

- **tagName:** Returns the tag name of the element (e.g., "DIV" for a <div> element).
- **id:** Returns the id attribute of the element.
- **className:** Returns the value of the class attribute.
- **href:** Returns the href attribute for anchor elements (<a>).
- **src:** Returns the src attribute for image elements (<img>).
- **innerHTML:** Returns or sets the HTML content of the element.

## 4. Events:

- JavaScript can be used to attach event listeners to DOM elements to respond to user interactions such as clicks, mouse movements, keyboard inputs, etc.
- Common events include click, mouseover, keydown, submit, and more. Event listeners can be added using addEventListener() and removed using removeEventListener().

## 5. Modifying DOM Elements:

- JavaScript can dynamically modify the content, structure, and style of HTML elements in response to user actions or other events.
- Elements can be created, removed, replaced, and manipulated as needed.

## 6. Security Considerations:

- Care should be taken when working with DOM to prevent cross-site scripting (XSS) attacks.
- Always sanitize and validate input and avoid injecting untrusted data into the DOM.

# JAVASCRIPT HTML DOM EVENTS

- JavaScript HTML DOM events are interactions and occurrences that happen in a web page, which can be detected and handled by JavaScript.
- Events can be triggered by user actions, such as mouse clicks, keyboard inputs, or changes in the document's state, and they provide a way to create dynamic and interactive web applications.

## Example:

### 1. Event Types:

- **User Events:** These events are triggered by user interactions, such as mouse clicks (click), mouse movements (mousemove), keyboard inputs (keydown, keyup), and form submissions.
- **Document Events:** These events are related to the state of the document, such as when the document is fully loaded (DOMContentLoaded), when it is completely loaded (load), or when it is unloaded (unload).
- **Focus Events:** These events are triggered when an element receives or loses focus, such as focus and blur events.
- **Form Events:** Events like change, input, and submit are related to HTML form elements and user inputs.
- **Media Events:** Events like play, pause, and ended are related to audio and video elements.
- **Network Events:** Events like online and offline are related to the network connection status.
- **Custom Events:** Developers can create custom events to handle specific application logic.

### 2. Event Handling:

- Event handling is the process of defining JavaScript functions (event handlers or listeners) that respond to specific events.
- Event listeners are attached to HTML elements using methods like addEventListener() to specify which events to listen for and what function to call when the event occurs.

### 3. Event Object:

- When an event occurs, an event object is created, which contains information about the event, such as the type of event, the target element, mouse coordinates, and more.
- Event object properties and methods can be used to access and manipulate event data.

### 4. Event Propagation:

- Events in the DOM follow a propagation model, which determines the order in which elements receive and handle events.
- There are two phases of event propagation: capturing phase and bubbling phase.
- Event propagation can be controlled using the event.stopPropagation() and event.preventDefault() methods.

### 5. Event Delegation:

- Event delegation is a technique where a single event listener is placed on a common ancestor of multiple elements.
- It allows you to handle events for dynamically created or numerous elements efficiently, as events bubble up to the ancestor element.

### 6. Removing Event Listeners:

- Event listeners should be removed when they are no longer needed to prevent memory leaks.
- You can remove event listeners using the removeEventListener() method.

### 7. Browser Compatibility:

- While DOM events are standardized, there may be variations in event handling across different browsers.
- Feature detection and cross-browser testing are essential to ensure compatibility.

## **8. Event-driven Programming:**

- JavaScript's event-driven nature is fundamental to modern web development and is used to create responsive and interactive user interfaces.

## **9. Common Use Cases:**

- Form validation and submission
- UI interactions like button clicks and menu selections
- Animations and transitions
- Real-time updates and data synchronization
- Handling user inputs and providing feedback
- Implementing drag-and-drop functionality

# **❖ DOM EVENTS IN JAVASCRIPT**

### **1. Mouse Events:**

- **click:** Triggered when an element is clicked.
- **dblclick:** Triggered on a double-click.
- **mouseover:** Triggered when the mouse enters an element.
- **mouseout:** Triggered when the mouse leaves an element.

### **2. Keyboard Events:**

- **keydown:** Triggered when a keyboard key is pressed down.
- **keyup:** Triggered when a keyboard key is released.
- **keypress:** Triggered when a character key is pressed down and released.

### **3. Form Events:**

- **submit:** Triggered when a form is submitted.
- **reset:** Triggered when a form is reset.
- **change:** Triggered when the value of a form element (input, select, etc.) changes.
- **input:** Triggered when the value of an input element changes (more responsive than change).
- **focus:** Triggered when an element receives focus.
- **blur:** Triggered when an element loses focus.
- **select:** Triggered when text is selected within an input or textarea element.

### **4. Window and Document Events:**

- **load:** Triggered when the document or a resource (e.g., image) finishes loading.
- **unload:** Triggered when the user navigates away from the page.
- **resize:** Triggered when the browser window is resized.
- **scroll:** Triggered when the user scrolls within an element.
- **DOMContentLoaded:** Triggered when the HTML document is fully loaded and parsed.
- **beforeunload:** Triggered before the user leaves the page (for confirmation).

### **5. Drag-and-Drop Events:**

- **dragstart:** Triggered when an element is dragged.
- **dragend:** Triggered when the element's drag operation is completed.
- **dragover:** Triggered when an element is being dragged over a valid drop target.
- **drop:** Triggered when an element is dropped onto a valid drop target.

### **6. Media Events:**



- **play:** Triggered when media (audio/video) starts playing.
- **pause:** Triggered when media is paused.
- **ended:** Triggered when media playback reaches the end.

## 7. Network Events:

- **online:** Triggered when the browser detects an internet connection.
- **offline:** Triggered when the browser loses its internet connection.

## 8. Custom Events:

- Developers can create and dispatch custom events using the CustomEvent constructor.
- **Touch Events** (for mobile and touch-enabled devices):
- **touchstart:** Triggered when a touch point is placed on the screen.
- **touchmove:** Triggered when a touch point moves along the screen.
- **touchend:** Triggered when a touch point is removed from the screen.

## ❖ DOM Mouse Events:

- **onclick / click:** Triggered when an element is clicked.
- **mouseover:** Triggered when the mouse pointer moves over an element.
- **mousemove:** Triggered when the mouse pointer moves within an element.
- **mouseout:** Triggered when the mouse pointer leaves an element.
- **dblclick:** Triggered when an element is double-clicked.

➤ [raj.html](#)

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Mouse Event </title>
<style>
.c{
    width: 200px; height: 200px;
    border: 2px solid black;
    text-align: center;
    box-sizing: border-box;
    padding-top: 30px; cursor: pointer;
}
.c1{
    background-color: aqua;
    margin: 10px;
    background-image: url(img1.jpg) ;
    background-size: cover;
}
.c2{
    background-color: red;
    background-image: url(img2.PNG) ;
    background-size: cover;
}
</style>
</head>
<body>
<button onclick="add()">Click</button>
```

```
<button onclick="add()">Click</button>
<button ondblclick="dadd()">Double Click</button>
<p></p>
<h1></h1>
<div class="c c1"> Mouserover</div>
<div class="c c2"> Mouseout</div>

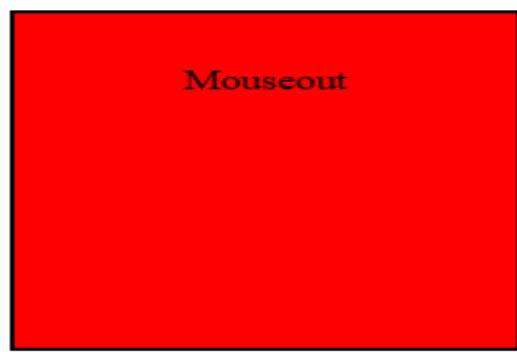
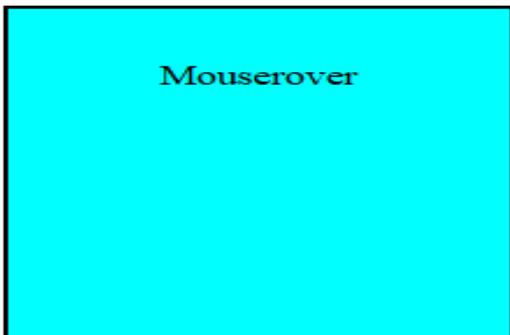
<script src="demo.js">
</script>

</body>
</html>
```

```
function add() { alert("Click button is successful!") }
function dadd() { alert("Double click successful!"); }
let div1 = document.querySelector(".c1");
let div2 = document.querySelector(".c2");
// div1.addEventListener("mouseover", () => {
//   console.log("user seeing my photo ")
//   let a = document.querySelector("p");
//   a.innerHTML = "You are visiting this div or my photo";
// });
div1.addEventListener("mousemove", () => {
  console.log("user seeing my photo ")
  let a = document.querySelector("p");
  a.innerHTML = "You are visiting this div or my photo";
});
div1.addEventListener("mouseout", () => {
  console.log("after 5 mintues you are out ")
  document.querySelector("p").innerHTML = "Cursor out";
});
div2.addEventListener("mouseout", () => {
  console.log("after 5 mintues you are out ")
  document.querySelector("p").innerHTML = "Cursor out";
});
```

**Click**    **Double Click**

**Cursor out**



```
<!DOCTYPE html>
<html>
<head>
  <title>Mouse Events Example</title>
  <style>
    .p{
      display: inline-flex;
      width: auto;
      height: auto;
    }
    #myElement1, #myElement2, #myElement3, #myElement4, #myElement5, #myElement6,
    #myElement7 {
      width: 100px;
      height: 100px;
      background-color: lightblue;
      text-align: center;
      line-height: 100px;
      cursor: pointer;
    }
  </style>
</head>
```

```
<body>
<div class="p">
<div id="myElement1">Click Me</div>
<div id="myElement2">Click Me</div>
<div id="myElement3">Click Me</div>
<div id="myElement4">Click Me</div>
<div id="myElement5">Click Me</div>
<div id="myElement6">Click Me</div>
<div id="myElement7">Click Me</div>
</div>
<script>
let myElement1 = document.getElementById("myElement1");
let myElement2 = document.getElementById("myElement2");
let myElement3 = document.getElementById("myElement3");
let myElement4 = document.getElementById("myElement4");
let myElement5 = document.getElementById("myElement5");
let myElement6 = document.getElementById("myElement6");
let myElement7 = document.getElementById("myElement7");
// Click event
myElement1.onclick = function () {
    myElement1.style.backgroundColor = "lightgreen";
    myElement1.textContent = "Clicked!";
}; // Mousedown event
myElement2.onmousedown = function () {
    myElement2.style.backgroundColor = "lightcoral";
    myElement2.textContent = "Mouse Down";
}; // Mouseup event
myElement3.onmouseup = function () {
    myElement3.style.backgroundColor = "lightblue";
    myElement3.textContent = "Mouse Up";
}; // Mousemove event
myElement4.onmousemove = function () {
    myElement4.style.backgroundColor = "lightpink";
}; // Mouseover event
myElement5.onmouseover = function () {
    myElement5.style.border = "2px solid red";
}; // Mouseout event
myElement6.onmouseout = function () {
    myElement6.style.border = "none";
}; // Double click event
myElement7.ondblclick = function () {
    myElement7.style.backgroundColor = "lightyellow";
    myElement7.textContent = "Double Clicked!";
};
</script>
</body>
</html>
```

Click Me Click Me Click Me Click Me Click Me Click Me Click Me



### Example:

#### ❖ keyboard events:

- **keydown**: triggers immediately when you press the key.
- **Keypress**: (if supported) triggers after keydown (only for printable characters).
- **keyup** triggers when you release the key.

### Key Differences at a Glance

Feature	keydown	keyup	keypress (Deprecated)
When Triggered	When the key is pressed down	When the key is released	After pressing a key (printable characters only)
Supports All Keys	Yes	Yes	No (only printable characters)
Recommended?	Yes	Yes	No (use <code>keydown</code> or <code>keyup</code> )

#### ➤ Raj.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Keyboard Navigation and Addition</title>
    <style>
        button {
            background-color: lightgray;
            border: 2px solid black;
            padding: 10px 20px;
            font-size: 16px;
            margin: 5px;
            transition: background-color 0.3s;
        }
        button:focus {
            background-color: green;
            outline: none;
        }
    </style>
</head>
<body>
    <h1>Keyboard Navigation Example</h1>
    <p>Use arrow keys to navigate. Press Enter to calculate the sum of two numbers.</p>
    <button type="button" id="d1">UpArrow</button>
    <button type="button" id="d2">DownArrow</button><br><br>
    <button type="button" id="d3">LeftArrow</button>
    <button type="button" id="d4">RightArrow</button><br><br>
```



```
<!-- Input fields for number addition -->
<input type="number" id="num1" placeholder="Enter first number">
<input type="number" id="num2" placeholder="Enter second number"><br><br>
<button type="button" id="calculateBtn">Press Enter to Add</button>
<p id="result"></p>
```

```
<script>
    // Function to handle focus changes
    function updateFocus(target) {
        document.querySelectorAll("button").forEach((btn) => {
            btn.style.backgroundColor = "lightgray";
        });
        target.style.backgroundColor = "red";
    }
    // Keyboard navigation for buttons
    document.addEventListener("keydown", (event) => {
        // Identify the currently focused element
        const focusedElement = document.activeElement; // built-in property
        // Switch cases to determine the navigation logic
        switch (event.key) {
            case "ArrowRight":
                if (focusedElement.id === "d1") {
                    // Move focus to d2
                    document.getElementById("d2").focus();
                    updateFocus(document.getElementById("d2")); // this used for color change
                } else if (focusedElement.id === "d3") {
                    document.getElementById("d4").focus();
                    updateFocus(document.getElementById("d4")); // this used for color change
                }
                break;
            case "ArrowLeft":
                if (focusedElement.id === "d2") {
                    document.getElementById("d1").focus();
                    updateFocus(document.getElementById("d1"));
                } else if (focusedElement.id === "d4") {
                    document.getElementById("d3").focus();
                    updateFocus(document.getElementById("d3"));
                }
                break;
            case "ArrowDown":
                if (focusedElement.id === "d1") {
                    document.getElementById("d3").focus();
                    updateFocus(document.getElementById("d3"));
                } else if (focusedElement.id === "d2") {
                    document.getElementById("d4").focus();
                    updateFocus(document.getElementById("d4"));
                }
        }
    })
}
```



```
break;  
case "ArrowUp":  
    if (focusedElement.id === "d3") {  
        document.getElementById("d1").focus();  
        updateFocus(document.getElementById("d1"));  
    } else if (focusedElement.id === "d4") {  
        document.getElementById("d2").focus();  
        updateFocus(document.getElementById("d2"));  
    }  
    break;  
case "Enter":  
    // Perform addition when Enter key is pressed  
    const num1 = parseFloat(document.getElementById("num1").value);  
    const num2 = parseFloat(document.getElementById("num2").value);  
    if (!isNaN(num1) && !isNaN(num2)) {  
        const result = num1 + num2;  
        document.getElementById("result").innerText = `Result: ${result}`;  
    } else {  
        document.getElementById("result").innerText = "Please enter valid numbers!";  
    }  
    break;  
}  
});  
</script>  
</body>  
</html>
```

## Keyboard Navigation Example

Use arrow keys to navigate between buttons. Press Enter to calculate the sum of two numbers.

UpArrow      DownArrow

LeftArrow      RightArrow

Enter first number      Enter second number

Press Enter to Add



## ❖ Form Events:

Name	Description
<code>submit</code>	Triggered when a form is submitted. Use it to handle or validate form data before submission.
<code>reset</code>	Triggered when a form is reset. Often used to perform actions when clearing form inputs.
<code>change</code>	Triggered when the value of a form element (input, select, checkbox, etc.) changes and loses focus.
<code>input</code>	Triggered in real-time when the value of an input or textarea changes.
<code>focus</code>	Triggered when a form element gains focus.
<code>blur</code>	Triggered when a form element loses focus.
<code>invalid</code>	Triggered when a form element does not meet its validation constraints (e.g., <code>required</code> ).
<code>select</code>	Triggered when text is selected in an input or textarea element.

### Example-1 submit and reset Event.

➤ Raj.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
</head>
<body>
  <form id="fsubmit">
    <label>Name </label>
    <input type="text" id="d1" required> <br><br>
    <label>Roll Number </label>
    <input type="text" id="d2" required> <br><br>
    <label>Mobile </label>
    <input type="tel" id="d3" maxlength="10" required> <br><br>
    <label>Email </label>
    <input type="email" id="d4" required> <br><br>
    <label>Branch: </label>
    <select id="bn">
      <option>CSE</option>
      <option>AIML</option>
      <option>CSIT</option>
    </select><br><br>
    <label>Gender</label>
    <input id="gn1" type="radio" name="gender" value="Male" required> Male
    <input id="gn2" type="radio" name="gender" value="Female"> Female
    <input id="gn3" type="radio" name="gender" value="Others"> Others
    <br><br>
    <button type="submit">Submit</button>
    <button type="reset">Reset</button>
  </form>
```

```
<script>
  document.getElementById("fsubmit").addEventListener("submit", function(event) {
    event.preventDefault(); // Prevent form submission and page reload
    // Collect form data
    const name = document.getElementById("d1").value;
    const rollNumber = document.getElementById("d2").value;
    const mobile = document.getElementById("d3").value;
    const email = document.getElementById("d4").value;
    const branch = document.getElementById("bn").value;
    // Get selected gender
    let gender = "";
    if (document.getElementById("gn1").checked) {
      gender = "Male";
    } else if (document.getElementById("gn2").checked) {
      gender = "Female";
    } else if (document.getElementById("gn3").checked) {
      gender = "Others";
    }
    // Store data in an object
    const studentData = {
      Name: name,
      RollNumber: rollNumber,
      Mobile: mobile,
      Email: email,
      Branch: branch,
      Gender: gender
    };
    // Display data using alert
    alert(`Student Information:\n
          Name: ${studentData.Name}\n
          Roll Number: ${studentData.RollNumber}\n
          Mobile: ${studentData.Mobile}\n
          Email: ${studentData.Email}\n
          Branch: ${studentData.Branch}\n
          Gender: ${studentData.Gender}`);
  });
  document.getElementById("fsubmit").addEventListener("reset", function() {
    alert("The form has been reset.");
  });
</script>
</body>
</html>
```



A watermark logo for 'DISCOVERY BHARAT' is centered over the page. It features a blue circular border with the word 'DISCOVERY' written along the top inner edge. Inside the circle, the letters 'BHARAT' are written vertically in a stylized font. Three yellow stars are scattered around the bottom right of the circle.

Name	<input type="text"/>
Roll Number	<input type="text"/>
Mobile	<input type="text"/>
Email	<input type="text"/>
Branch:	<input type="text" value="CSE"/>
Gender	<input type="radio"/> Male <input type="radio"/> Female <input type="radio"/> Others
<input type="button" value="Submit"/> <input type="button" value="Reset"/>	

### Example-2: Input event and change event

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Change and Input Events</title>
</head>
<body>
    <h2>Change and Input Events Example</h2>
    <form id="simpleForm">
        <label for="username">Name:</label>
        <input type="text" id="username" value="rid bharat">
        <p id="nameDisplay">Name: </p> <br><br>
        <label for="age">Age:</label>
        <input type="number" id="age" value="20">
        <p id="ageDisplay">Age: </p> <br><br>
        <label for="gender">Gender:</label>
        <select id="gender">
            <option value="">Select Gender</option>
            <option value="Male">Male</option>
            <option value="Female">Female</option>
            <option value="Other">Other</option>
        </select>
        <p id="genderDisplay">Gender: </p> <br><br>
        <button type="submit">Submit</button>
    </form>
    <script>
        // Handling the input event for the Name field
        document.getElementById("username").addEventListener("input", function () {
            document.getElementById("nameDisplay").innerText = "Name: " + this.value;
        });

        // Handling the change event for the Age field
        document.getElementById("age").addEventListener("change", function () {
            document.getElementById("ageDisplay").innerText = "Age: " + this.value;
        });

        // Handling the change event for the Gender field
        document.getElementById("gender").addEventListener("change", function () {
            document.getElementById("genderDisplay").innerText = "Gender: " + this.value;
        });

        // Handling the submit event on the form
        document.getElementById("sf").addEventListener("submit", function (event) {
            event.preventDefault();
            alert("Form submitted successfully!");
        });
    </script>
</body>
</html>
```

### Change and Input Events Example

Name: rid bharat

Name: rid bharat

Age: 20

Age: 20

Gender: Male

Gender: Male

Submit

### Example-3: Input event and change event

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Handling</title>
</head>
<body>
    <h2>Event Handling Example</h2>
    <form>
        <label for="username">Username:</label>
        <input type="text" id="username" required>
        <br><br>
        <label for="age">Age:</label>
        <input type="number" id="age" min="18" max="100">
        <br><br>
        <select id="color">
            <option value="">Select a color</option>
            <option value="red">Red</option>
            <option value="green">Green</option>
            <option value="blue">Blue</option>
        </select>
        <br><br>
        <p id="result"></p>
    </form>
    <script>
        // Focus event
        document.getElementById("username").addEventListener("focus", function() {
            document.getElementById("result").innerText = "Username field is focused";
        });
        // Invalid event
        document.getElementById("username").addEventListener("invalid", function() {
            document.getElementById("result").innerText = "Please enter a valid username";
        });
        // Select event
        document.getElementById("color").addEventListener("change", function() {
            var selectedColor = this.value;
            document.getElementById("result").innerText = "You selected: " + selectedColor;
        });
    </script>
</body>
</html>
```

### Event Handling Example

Username:

Age:

You selected: red



## ❖ Window and Document Events:

- 1) **load**: Triggered when a resource or webpage fully loads.
- 2) **unload**: Triggered when a webpage is being unloaded or closed.
- 3) **beforeunload**: Triggered before the webpage is unloaded, allowing prompts or warnings.
- 4) **scroll**: Triggered when the user scrolls within an element or the webpage.
- 5) **resize**: Triggered when the browser window or an element is resized.

### Example: load Event:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Load Event Example</title>
</head>
<body>
<h1>Welcome to My Page!</h1>
<p id="status">Loading content...</p>
<p id="time"></p>
<script>
window.addEventListener('load', () => {
  document.getElementById('status').textContent = "Page Loaded Successfully!";
  const now = new Date();
  document.getElementById('time').textContent = `Current Time: ${now.toLocaleTimeString()}`;
});
</script>
</body>
</html>
```

## Welcome to My Page!

Page Loaded Successfully!

Current Time: 9:15:47 PM

### Example: unload and beforeunload events

Feature	beforeunload	unload
When It Triggers	Before the page is about to unload.	After the page has been unloaded.
Purpose	Warn the user or prevent navigation.	Perform cleanup tasks.
Allows Prevent Navigation?	Yes, using <code>event.returnValue</code> .	No. Navigation cannot be stopped.
Dialog/Message	Can trigger a browser confirmation dialog.	No dialog or message is possible.
Use Case	Warn about unsaved changes or confirm exit.	Send data or clean up connections.
Browser Restrictions	Custom messages in dialogs are ignored.	Often limited to asynchronous tasks.



**Example:**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Unload Event Example</title>
</head>
<body>
<h1>Stay on this page!</h1>
<p id="leaveMessage">Don't leave without checking out all the content!</p>
<script>
window.addEventListener('beforeunload', (event) => {
  event.preventDefault();
  event.returnValue = '';
  document.getElementById('leaveMessage').textContent =
    'You attempted to leave the page at ' + new Date().toLocaleTimeString();
});
</script>
</body>
</html>
```

The screenshot shows a web browser window with the URL 127.0.0.1:5500/index3.html. The main content area displays the text "Stay on this page!" and "Don't leave without checking out all the content!". A blue circular watermark with the text "INNOVATION DISCOVERY DEDICATION" is overlaid on the background. A modal dialog box is open in the upper right corner, asking "Leave site?" with the message "Changes you made may not be saved." It contains two buttons: "Leave" (in blue) and "Cancel" (in light blue).

The screenshot shows a web browser window with the URL 127.0.0.1:5500/index3.html. The main content area displays the text "Stay on this page!" and "You attempted to leave the page at 9:40:09 PM". A blue circular watermark with the text "INNOVATION DISCOVERY DEDICATION" is overlaid on the background. The "Leave" button from the previous dialog has been clicked, and the browser has navigated away from the page.

## Example: Scroll and resize:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Scroll Event Example</title>
<style>
body {
  height: 1500px; /* To make the page scrollable */
  margin: 0;
  font-family: Arial, sans-serif;
  text-align: center;
  transition: background-color 0.5s ease;
}
h1 {
  position: fixed;
  top: 20px;
  left: 50%;
  transform: translateX(-50%);
  background: rgba(94, 71, 71, 0.8);
  padding: 10px 20px;
  border-radius: 5px;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}
</style>
</head>
<body>
<h1 id="scrollInfo">Scroll to change the background color!</h1>
<script>
window.addEventListener('scroll', () => {
  const scrollY = window.scrollY; // Get vertical scroll position
  const colorValue = Math.min(255, scrollY / 5); // Calculate color intensity based on scroll
  document.body.style.backgroundColor = `rgb(${colorValue}, ${255 - colorValue}, ${150})`;
  document.getElementById('scrollInfo').textContent = `Scroll position: ${scrollY}px`;
});
</script>
</body></html>
```

### How It Works:

- As the user scrolls down, the background color gradually changes.
- The RGB values are dynamically adjusted based on the scroll position.
- The scroll position is displayed in the heading.

Scroll to change the background color!

Scroll position: 0px

Scroll position: 948.1818237304688px



### Example of Resize:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Resize Event Example</title>
<style>
body {
    font-family: Arial, sans-serif;
    text-align: center;
    margin: 0;
}
.box {
    margin: 20px auto;
    width: 50vw;
    height: 30vh;
    border: 2px dashed darkblue;
    display: flex;
    align-items: center;
    justify-content: center;
    background: lightcyan;
    transition: all 0.3s ease;
}
.info {
    font-size: 18px;
    font-weight: bold;
    color: darkblue;
}
</style>
</head>
<body>
<h1>Resize the Window!</h1>
<div class="box">
<p class="info" id="boxInfo">Box Size: Loading...</p>
</div>
<script>
const updateBoxSize = () => {
    const box = document.querySelector('.box');
    const boxWidth = box.offsetWidth; //Get the current width of the box
    const boxHeight = box.offsetHeight; //Get the current height of the box
    document.getElementById('boxInfo').textContent = `Box Size: ${boxWidth}px x ${boxHeight}px`;
};
// Initial size update
updateBoxSize();
// Add event listener for resize
window.addEventListener('resize', updateBoxSize);
</script>
</body>
</html>
```

## Resize the Window!

Box Size: 625px x 169px

## Resize the Window!

Box Size: 625px x 169px

## ❖ Drag-and-Drop Events:

- 1) **dragstart**: Triggered when the user starts dragging an element.
- 2) **dragend**: Triggered when the user stops dragging an element.
- 3) **dragover**: Triggered when a dragged element is over a valid drop target.
- 4) **drop**: Triggered when a dragged element is dropped onto a valid drop target.

➤ **raj.html**:

```
<!DOCTYPE html>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Drag and Drop Example</title>
<style>
body { font-family: Arial, sans-serif; margin: 10px; }
#da { border: 2px dashed #007bff; padding: 10px; text-align: center; min-height: 100px; width: 400px; }
img { width: 200px; cursor: grab; }
#dm { margin-top: 20px; font-size: 1.2em; }
#sm { margin-top: 10px; color: #007bff; font-size: 1.1em; }
</style>
</head>
<body>
<h1>Drag and Drop Example</h1>
<p>Drag the image into the dashed box to see the effect.</p>
<div id="da">Drop the image here</div>

<div id="sm"></div>
<div id="dm"></div>
<script>
const dragImage = document.getElementById('di');
const dragArea = document.getElementById('da');
const startMessage = document.getElementById('sm');
const dropMessage = document.getElementById('dm');

let droppedImage = null; // Variable to store the dropped image
// Handle dragstart (on the image)
dragImage.addEventListener('dragstart', () => {
  startMessage.textContent = 'You started dragging the image!';
  startMessage.style.color = '#007bff';
});

// Handle dragend (on the image)
dragImage.addEventListener('dragend', () => {
  startMessage.textContent = 'You stopped dragging the image!';
  startMessage.style.color = '#28a745';
});
```



```
// Handle dragover (on the drop area)
dragArea.addEventListener('dragover', (event) => {
  event.preventDefault();
});

// Handle drop (on the drop area)
dragArea.addEventListener('drop', (event) => {
  event.preventDefault();
  droppedImage = dragImage.src; // Store the image source in the variable
  dropMessage.textContent = `Image successfully dropped! Image source: ${droppedImage}`;
  dropMessage.style.color = '#28a745';
});
</script>
</body>
</html>
```

## Drag and Drop Example

Drag the image into the dashed box to see the effect.

Drop the image here



You stopped dragging the image!

Image successfully dropped! Image source: <http://127.0.0.1:5500/img3.jpg>

### Task

#### Drag and Drop Example

Drag the image into the dashed box to see the effect.

Drop the image here



You stopped dragging the image!

Image successfully dropped! Image source: <http://127.0.0.1:5500/img3.jpg>

Show Dropped Image



```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Drag and Drop Example</title>
<style>
body { font-family: Arial, sans-serif; margin: 10px; }
#da { border: 2px dashed #007bff; padding: 10px; text-align: center; min-height: 100px; width: 400px; }
img { width: 200px; cursor: grab; }
#dm { margin-top: 20px; font-size: 1.2em; }
#sm { margin-top: 10px; color: #007bff; font-size: 1.1em; }
#imageContainer { margin-top: 20px; }
</style>
</head>
<body>
<h1>Drag and Drop Example</h1>
<p>Drag the image into the dashed box to see the effect.</p>
<div id="da">Drop the image here</div>

<div id="sm"></div>
<div id="dm"></div>
<button id="showImageBtn">Show Dropped Image</button>
<div id="imageContainer"></div>
<script>
const dragImage = document.getElementById('di');
const dragArea = document.getElementById('da');
const startMessage = document.getElementById('sm');
const dropMessage = document.getElementById('dm');
const showImageBtn = document.getElementById('showImageBtn');
const imageContainer = document.getElementById('imageContainer');
let droppedImage = null; // Variable to store the dropped image // Handle dragstart (on the image)
dragImage.addEventListener('dragstart', () => {
  startMessage.textContent = 'You started dragging the image!';
  startMessage.style.color = '#007bff';
}); // Handle dragend (on the image)
dragImage.addEventListener('dragend', () => {
  startMessage.textContent = 'You stopped dragging the image!';
  startMessage.style.color = '#28a745';
}); // Handle dragover (on the drop area)
dragArea.addEventListener('dragover', (event) => {
  event.preventDefault();
}); // Handle drop (on the drop area)
dragArea.addEventListener('drop', (event) => {
  event.preventDefault();
  droppedImage = dragImage.src; // Store the image source in the variable
  dropMessage.textContent = `Image successfully dropped! Image source: ${droppedImage}`;
  dropMessage.style.color = '#28a745';
});
```

```
// Show dropped image when the button is clicked
showImageBtn.addEventListener('click', () => {
  if (droppedImage) {
    const img = document.createElement('img');
    img.src = droppedImage;
    img.alt = 'Dropped Image';
    img.style.width = '200px';
    imageContainer.innerHTML = ""; // Clear previous images
    imageContainer.appendChild(img); // Display the dropped image
  } else {
    imageContainer.innerHTML = 'No image dropped yet!';
  }
});
</script>
</body>
</html>
```

## Drag and Drop Example

Drag the image into the dashed box to see the effect.

Drop the image here



Show Dropped Image

No image dropped yet!



## ❖ DOM Media Events:

- **play**: Triggered when the video or audio starts playing.
- **pause**: Triggered when the video or audio is paused.
- **ended**: Triggered when the video or audio playback finishes.

**raj.html:**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Video Events with Skip Buttons</title>
</head>
<body>
    <h1>Video Player Example</h1>
    <!-- Video Element -->
    <video id="myVideo" width="480">
        <source id="videoSource" src="abv2.mp4" type="video/mp4">
    </video>
    <!-- Status Display -->
    <p id="status">Status: Video is not playing</p>
    <!-- Buttons -->
    <button id="playBtn">Play</button>
    <button id="pauseBtn">Pause</button>
    <button id="endBtn">End</button>
    <button id="downloadBtn">Download</button>
    <button id="leftBtn">⬅ Back 5 Seconds</button>
    <button id="rightBtn">➡ Forward 5 Seconds</button>
    <!-- Script for Event Handlers -->
    <script>
        const video = document.getElementById('myVideo');
        const videoSource = document.getElementById('videoSource'); // Get the video source
        const status = document.getElementById('status');
        const playBtn = document.getElementById('playBtn');
        const pauseBtn = document.getElementById('pauseBtn');
        const endBtn = document.getElementById('endBtn');
        const downloadBtn = document.getElementById('downloadBtn');
        const leftBtn = document.getElementById('leftBtn'); // Backward Button
        const rightBtn = document.getElementById('rightBtn'); // Forward Button
        // Play Button Event
        playBtn.addEventListener('click', () => {
            video.play();
            status.textContent = 'Status: Video is playing';
        });
        // Pause Button Event
        pauseBtn.addEventListener('click', () => {
            video.pause();
            status.textContent = 'Status: Video is paused';
        });
    </script>
```



```
// End Button Event
endBtn.addEventListener('click', () => {
  video.currentTime = video.duration;
  status.textContent = 'Status: Video has ended';
});

// Download Button Event
downloadBtn.addEventListener('click', () => {
  const videoURL = videoSource.src; // Get the video source URL
  const anchor = document.createElement('a'); // Create a temporary anchor element
  anchor.href = videoURL;
  anchor.download = 'video.mp4'; // Set the download filename
  anchor.click(); // Trigger the download
});

// Backward Button Event
leftBtn.addEventListener('click', () => {
  video.currentTime = Math.max(0, video.currentTime - 5);
});

// Skip 5 seconds back, ensure it doesn't go below 0
status.textContent = `Status: Video skipped back 5 seconds. Current time: ${Math.floor(video.currentTime)}s`;
};

// Forward Button Event
rightBtn.addEventListener('click', () => {
  video.currentTime = Math.min(video.duration, video.currentTime + 5);
});

// Skip 5 seconds forward, ensure it doesn't exceed video duration
status.textContent = `Status: Video skipped forward 5 seconds. Current time: ${Math.floor(video.currentTime)}s`;
};

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

## Video Player Example



Status: Video skipped forward 5 seconds. Current time: 10s



## ❖ Network Events:

- **Online:** Triggered when the device regains internet connection.
- **Offline:** Triggered when the device loses internet connection.

```
raj.html      <!DOCTYPE html>
              <html lang="en">
              <head>
                <title>Network Events Example</title>
                <style>
                  #status {
                    font-size: 20px;
                    font-weight: bold;
                    color: white;
                    padding: 10px;
                    border-radius: 5px;
                    display: inline-block;
                  }
                  .online { background-color: green }
                  .offline { background-color: red; }
                </style>
              </head>
              <body>
                <h1>Network Status Example</h1>
                <p id="status" class="online">You are online</p>
                <script>
                  const statusElement = document.getElementById('status');
                  // Function to update the status
                  const updateStatus = () => {
                    if (navigator.onLine) {
                      statusElement.textContent = 'You are online';
                      statusElement.className = 'online';
                    } else {
                      statusElement.textContent = 'You are offline';
                      statusElement.className = 'offline';
                    }
                  };
                  // Event listeners for network events
                  window.addEventListener('online', updateStatus);
                  // Triggered when the user goes online
                  window.addEventListener('offline', updateStatus);
                  // Triggered when the user goes offline
                  // Initial status check
                  updateStatus();
                </script>
              </body> </html>
```

### Network Status Example

You are online

### Network Status Example

You are offline



❖ **Custom Events:**

- Developers can create and dispatch custom events using the CustomEvent constructor.
- Touch Events (for mobile and touch-enabled devices):
  - touchstart:
  - touchmove:
  - touchend:

➤ **raj.html:**

```
<!DOCTYPE html>
<html>
<head>
<title>Custom Events and Touch Events Example</title>
<style>
/* Add your CSS styles here */
body {
    font-family: Arial, sans-serif;
    text-align: center;
}
#custom-event-section {
    padding: 20px;
    border: 1px solid #ccc;
    margin-bottom: 20px;
}
#touch-event-section {
    padding: 20px;
    border: 1px solid #ccc;
    margin-bottom: 20px;
}
</style>
</head>
<body>
<h1>Custom Events and Touch Events Example</h1>
<!-- Custom Events Example -->
<div id="custom-event-section">
<h2>Custom Events Example</h2>
<button id="custom-event-button">Click Me</button>
<div id="custom-event-output"></div>
</div>
<!-- Touch Events Example -->
<div id="touch-event-section">
<h2>Touch Events Example</h2>
<div id="touch-event-output"></div>
</div>
<script>
// Custom Events Example
Const customEventButton = document.getElementById('custom-event-button');
Const customEventOutput = document.getElementById('custom-event-output');

// Create a custom event

```

```
constcustomEvent = new Event('customEvent');
customEventButton.addEventListener('click', function () {
    // Dispatch the custom event when the button is clicked
    customEventOutput.innerText = 'Custom Event: Button clicked!';
    customEventButton.dispatchEvent(customEvent);
});
// Listen for the custom event and handle it
customEventButton.addEventListener('customEvent', function () {
    customEventOutput.innerText = 'Custom Event: Custom event triggered!';
});
// Touch Events Example
consttouchEventOutput = document.getElementById('touch-event-output');
// Function to handle touchstart event
function handleTouchStart(event) {
    touchEventOutput.innerText = 'Touch Start Event: Touch detected.';
}
// Function to handle touchmove event
function handleTouchMove(event) {
    touchEventOutput.innerText = 'Touch Move Event: Touch moving.';
}
// Function to handle touchend event
function handleTouchEnd(event) {
    touchEventOutput.innerText = 'Touch End Event: Touch ended.';
}
// Add touch event listeners
touchEventOutput.addEventListener('touchstart', handleTouchStart);
touchEventOutput.addEventListener('touchmove', handleTouchMove);
touchEventOutput.addEventListener('touchend', handleTouchEnd);
</script>
</body>
</html>
```

## Custom Events and Touch Events Example

### Custom Events Example

Click Me

### Touch Events Example

# Regular expressions in JavaScript

- regular expression (regex)\* is a sequence of characters that forms a search pattern. It is used to find, match, or replace patterns in strings. Regular expressions are powerful tools for string manipulation, including tasks like validation, extraction, and replacement of text.

## 1. Matching Digits:

- Match any digit: \d
- Match any non-digit: \D

## 2. Matching Words and Characters:

- Match any word character (alphanumeric plus underscore): \w
- Match any non-word character: \W

## 3. Matching Whitespace:

- Match any whitespace character: \s
- Match any non-whitespace character: \S

## 4. Matching Specific Characters:

- Match a specific character (e.g., a): a
- Match a character range (e.g., a to z): [a -z]
- Match any character except those in a range: `[^0-9]` (matches anything that is not a digit)

## 5. Quantifiers (Repetition):

- Match zero or more: \*
- Match one or more: +
- Match zero or one: ?
- Match exactly n times: {n}
- Match n or more times: {n,}
- Match between n and m times: {n,m}

## 6. Anchors:

- Match the start of a line or string: ^
- Match the end of a line or string: \$

## 7. Grouping and Capturing:

- Create a capturing group: (pattern)
- Refer to captured groups: '\1', '\2', etc.

## 8. Alternation:

- Match one of several patterns: `pattern1|pattern2`

## 9. Word Boundaries:

- Match at the beginning of a word: \b
- Match at the end of a word: \B

## 10. Quantifier Shortcuts:

- Match exactly one digit: '\d{1}'
- Match two digits: '\d{2}'
- Match between 3 and 5 word characters: '\w{3,5}'

## 11. Email Validation:

- Basic email validation (not exhaustive): `^@[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}\$`

## 12. URL Validation:

- Basic URL validation (not exhaustive): `^(https?|ftp)://[^\\s/\$.?#].[^\\s]\*\$`

## ❖ Types of method.

- There are following types of method in regular expression:

1. `test()`
2. `match()`
3. `matchAll()`
4. `replace()`
5. `replaceAll()`
6. `exec()`
7. `search()`
8. `split()`

### 1. `test()` Method

- **Description:** Tests whether a regular expression matches a string. Returns true or false.
- **Syntax:** `regex.test(string)`

#### 1. Test for a Word

```
let regex = /hello/;  
let str1 = "hello world";  
let str2 = "Hi there!";  
console.log(regex.test(str1)); // Output: true  
console.log(regex.test(str2)); // Output: false
```

**Explanation:** Checks if word "hello" is present in the string. Returns true for str1 but false for str2.

#### 2. Case-Insensitive Matching

```
let regex = /hello/i; // `i` makes it case-insensitive  
let str1 = "HELLO world";  
let str2 = "Goodbye!";  
console.log(regex.test(str1)); // Output: true  
console.log(regex.test(str2)); // Output: false
```

**Expln:** i flag makes search case-insensitive. It matches "HELLO" in str1 but finds nothing in str2.

#### 3. Test for Digits

```
let regex = /\d+/; // Matches one or more digits  
let str1 = "I have 2 apples";  
let str2 = "No numbers here";  
console.log(regex.test(str1)); // Output: true  
console.log(regex.test(str2)); // Output: false
```

**Exp:** \d+ checks for one or more digits. It matches number 2 in str1 but finds no digits in str2.

#### 4. Check for Start of String

```
let regex = /^JavaScript/; // `^` ensures the string starts with "JavaScript"  
let str1 = "JavaScript is fun";  
let str2 = "I love JavaScript";  
console.log(regex.test(str1)); // Output: true  
console.log(regex.test(str2)); // Output: false
```

**Explanation:** ^ ensures that "JavaScript" is at the beginning of the string. It matches str1 but not str2.

#### 5. Test for Whitespace

```
let regex = /\s/; // Matches any whitespace character (space, tab, etc.)  
let str1 = "Hello World";
```



```
let str2 = "NoSpacesHere";
console.log(regex.test(str1)); // Output: true
console.log(regex.test(str2)); // Output: false
```

**Explanation:** The \s matches any whitespace. It detects the space between "Hello" and "World" in str1 but finds no spaces in str2.

### ❖ Flags (Modifiers)

- 1) **i**: Case-insensitive matching
- 2) **g**: Global match (find all matches)
- 3) **m**: Multi-line matching

**1. i - Case-Insensitive Matching** i flag makes search case-insensitive, meaning it doesn't differentiate between uppercase and lowercase letters.

**Example:**

```
let regex = /hello/i; // Case-insensitive matching
let str1 = "HELLO world!";
let str2 = "Hi, Hello!";
let str3 = "No match here.";
console.log(regex.test(str1)); // Output: true (matches "HELLO")
console.log(regex.test(str2)); // Output: true (matches "Hello")
console.log(regex.test(str3)); // Output: false
```

### 2. g - Global Match

- The g flag allows you to find all matches in a string instead of stopping after the first match. Although test() does not return all matches (only true or false), you can see the flag's behavior.

**Example:** let regex = /hello/g; // Global match

```
let str1 = "hello world! hello again!";
let str2 = "No match here.";
console.log(regex.test(str1)); // Output: true (finds "hello")
console.log(regex.test(str2)); // Output: false (no match)
```

### 3. m - Multi-Line Matching

- m flag allows the ^ (start of line) and \$ (end of line) anchors to work across multiple lines.

**Example:**

```
let regex = /^hello/m; // Match "hello" at the start of any line
let str1 = "hello world!\nHello again!";
let str2 = "Hi there!\nhello";
console.log(regex.test(str1)); // Output: true (matches "hello" in the first line)
console.log(regex.test(str2)); // Output: true (matches "hello" in the second line)
```

### Combined Example: i, g, and m Flags

- You can combine flags to use multiple functionalities together.

```
let regex = /hello/img; // Case-insensitive, global, and multi-line match
let str = "HELLO world!\nhello again!\nHi, Hello!";
console.log(regex.test(str)); // Output: true (matches "HELLO", "hello" across lines)
```



## 2. match() method

- match() method in JavaScript is used to extract matches for a regular expression from a string. It returns an **array** of matches or null if no match is found.

### Example:

#### 1. Simple Match

```
let str = "I love JavaScript!";
let regex = /love/;
let result = str.match(regex);
console.log(result);

// Output: ['love']
// [ 'love', index: 2, input: 'I love JavaScript!', groups: undefined ]
```

Example with a Named Capturing Group:

```
let str = "I love JavaScript!";
let regex = /(<emotion>love)/;
```

// Named capturing group

```
let result = str.match(regex);
console.log(result);
```

**Output:**

```
[ 'love',
'love',
index: 2,
input: 'I love JavaScript!',
groups: { emotion: 'love' }
]
```

### Explanation:

- 'love':**  
This is the part of the string that matched the regular expression /love/.
- index: 2:**  
The starting position of the match in the string. In this case, the word 'love' starts at the 2nd index (0-based indexing) in the string "I love JavaScript!".
- input: 'I love JavaScript!':**  
The full input string that was searched.
- groups: undefined:**
  - The groups property is used to store any **named capturing groups** from the regular expression.
  - In your example, the regular expression /love/ does not have any capturing groups (e.g., (?<name>...)), so groups is undefined.
  - If you had a named capturing group, it would contain an object with the group names and their matched values.

## 2. Case-Insensitive Matching

```
let str = "I LOVE JavaScript!";
let regex = /love/i; // `i` makes it case-insensitive
let result = str.match(regex);
console.log(result); // Output: ['LOVE']
```

**Explanation:** The i flag allows the regex to match "LOVE" regardless of case.

## 3. Global Match

```
let str = "I love JavaScript! I also love Python.";
let regex = /love/g; // `g` finds all matches
let result = str.match(regex);
console.log(result); // Output: ['love', 'love']
```

**Explanation:** The g flag returns all occurrences of "love" as an array.

## 4. Match Digits

```
let str = "I have 2 apples and 3 bananas.";
let regex = /\d+/g; // '\d+' matches one or more digits
let result = str.match(regex);
console.log(result); // Output: ['2', '3']
```

**Explanation:** The regex matches all numbers (2 and 3) in the string.

## 5. Match Words at the Start

```
let str = "JavaScript is fun!";
let regex = /^JavaScript/; // `^` ensures the match starts at the beginning
let result = str.match(regex);
console.log(result); // Output: ['JavaScript']
```

**Explanation:** The regex matches "JavaScript" only if it's at the start of the string. **Return Value:**

### 1. If a match is found:

- Returns an array:
- The first element is the full match.
- The remaining elements contain captured groups (if any).

### 2. If no match is found:

Returns null.

#### Example of No Match:

```
let str = "I love coding.";
let regex = /python/;
let result = str.match(regex);
console.log(result); // Output: null
```

## 3. matchAll() Method

- **matchAll()** method in JavaScript returns an iterator of all matches of a regular expression in a string, including detailed information about each match like capturing groups and positions. It requires the **global flag (g)** in the regex.

#### Syntax: string.matchAll(regex)

- **string:** The string to search within.
- **regex:** The regular expression to match against, which must include the g (global) flag.

#### Example: Find All Numbers in a String

```
let str = "I have 2 apples, 3 bananas, and 10 oranges.";
let regex = /\d+/g; // Matches all numbers
let matches = str.matchAll(regex);
for (const match of matches) {
  console.log(match[0]); // The matched number
}
Output: 2, 3, 10
```

#### Example 1: Find All Vowels in a String

```
let str = "Hello, how are you?";
let regex = /[aeiou]/g; // Matches vowels (a, e, i, o, u)
let matches = str.matchAll(regex);
for (const match of matches) {
  console.log(match[0]); // The matched vowel
}
Output: e, o, o, a, e, o
```

#### Example 2: Find All Words in a Sentence

```
let str = "JavaScript is fun!";
let regex = /\b\w+\b/g; // Matches complete words
let matches = str.matchAll(regex);
for (const match of matches) {
  console.log(match[0]); // The matched word
}
```

**Output:** JavaScript  
is  
fun

#### Example 1:

- The regular expression /[aeiou]/g looks for all vowels in the string.
- The loop prints each vowel one by one.

#### Example 2:

- The regular expression /\b\w+\b/g matches entire words in the string.
- \b ensures it matches word boundaries, and \w+ matches one or more word characters.

Aspect	<code>match()</code>	<code>matchAll()</code>
Output	Returns an array of matches or <code>null</code> .	Returns an iterator with detailed match info.
Multiple Matches	Needs <code>g</code> flag to return all matches.	Always returns all matches.
Details Provided	No capturing groups or match index.	Includes match index, input, and groups.
Use Case	When only matched strings are needed.	When detailed match info is required.

## 4. `replace()` Method

- `replace()` method is used to replace a specific substring or pattern (defined by a regular exp..) in a string with a new string. It replaces only **first match** unless `global` flag (`g`) is used.

### Syntax: `string.replace(pattern, replacement)`

- `pattern`: The substring or regular expression to match.
- `replacement`: The string or function to replace the matched substring.

**Example 1:** Replacing a Word Replace the word "apple" with "orange" in a string.

```
let str = "I like apple pie.";
let newStr = str.replace(/apple/, "orange");
console.log(newStr); // Output: "I like orange pie."
```

**Explanation:** This replaces the first occurrence of "apple" with "orange".

**Example 2:** Replacing All Occurrences (Global Match) Use the `g` flag to replace all occurrences of "apple" with "orange".

**Example:**

```
let str = "I have an apple and another apple.";
let newStr = str.replace(/apple/g, "orange");
console.log(newStr); // Output: "I have an orange and another orange."
```

**Explanation:** The `g` flag ensures that both occurrences of "apple" are replaced.

**Example 3:** Case-Insensitive Replacement Use the `i` flag to replace "Apple" (case-insensitive) with "orange".

**Example:**

```
let str = "I love Apple pie!";
let newStr = str.replace(/apple/i, "orange");
console.log(newStr); // Output: "I love orange pie!"
```

**Explanation:** The `i` flag makes the search case-insensitive, so it matches "Apple" regardless of case.

**Example 4:** Replacing Digits with a Symbol Replace any number in the string with the # symbol.

**Example:** let str = "My house number is 1234.";

```
let newStr = str.replace(/\d+/g, "#");
console.log(newStr); // Output: "My house number is #."
```

**Explanation:** The regular expression `\d+` matches any number, and the `g` flag ensures that all numbers in the string are replaced.

**Example 5:** Removing Extra Spaces Remove multiple spaces between words and replace them with a single space.

**Example:** let str = "This is a sentence.";

```
let newStr = str.replace(/\s+/g, " ");
console.log(newStr); // Output: "This is a sentence."
```

**Explanation:** The regular expression `\s+` matches one or more whitespace characters, and the `g` flag ensures that all extra spaces are replaced with a single space.



## 5. replaceAll() Method

- **replaceAll()** method replaces **all occurrences** of a substring or pattern in a string with a new string. Unlike replace(), it does not require the global flag (g) and replaces all matches by default.

**Syntax:** `string.replaceAll(pattern, replacement)`

**Example: Replacing All Occurrences of a Word:**

```
let str = "I like apples, apples are tasty.";
let result = str.replaceAll("apples", "oranges");
console.log(result); // Output: "I like oranges, oranges are tasty."
```

**Example: Replacing Using Regular Expression:**

```
let str = "1, 2, 3, 4";
let result = str.replaceAll(/\d/g, "#");
console.log(result); // Output: "#, #, #, #"
```

**Example: Replacing All Spaces:**

```
let str = "Hello World! How are you?";
let result = str.replaceAll(" ", "-");
console.log(result); // Output: "Hello-World!-How-are-you?"
```

### Key Differences Between `replace()` and `replaceAll()`:

Aspect	<code>replace()</code>	<code>replaceAll()</code>
Default Behavior	Replaces the first occurrence only (unless <code>g</code> is used).	Replaces all occurrences by default.
Regex Requirement	Can use a regex with the <code>g</code> flag.	No need for the <code>g</code> flag to replace all.
Availability	Available in most browsers.	Available in modern browsers only.

## 6. exec() Method

The exec() method executes a regular expression on a string and returns the first match as an array with detailed information (e.g., index, input). If no match is found, it returns null.

**Syntax:** `regex.exec(string)`

**Example-1: Find a Word**

```
let regex = /cat/;
let str = "The cat is on the mat.";
let result = regex.exec(str);
console.log(result);
```

**// Output:** [ 'cat', index: 4, input: 'The cat is on the mat.' ]

**Example-2: Find Numbers**

```
let regex = /\d+/";
let str = "There are 123 apples.";
console.log(regex.exec(str));
```

**// Output:** [ '123', index: 10, input: 'There are 123 apples.' ]

**Example-3: No Match**

```
let regex = /cat/;
let str = "The dog is barking.";
console.log(regex.exec(str));
// Output: null
```

## 7. search() Method

The search() method searches for a match of a regular expression in a string and returns the index of the first match. If no match is found, it returns -1.

**Syntax:** string.search(regex)

**Example-1: Find the First Match**

```
let str = "I love JavaScript!";  
let index = str.search(/JavaScript/);  
console.log(index); // Output: 7
```

**Example 2: Find the First Match**

```
let str = "I love rid bharat rid patna !";  
console.log(str.search(/rid/)); // Output: 2
```

**Example 3: Case-Insensitive Search**

```
let str = "I like Cats.";  
console.log(str.search(/cats/i)); // Output: 7
```

**Example 3: No Match Found**

```
let str = "Hello world!";  
console.log(str.search(/dog/)); // Output: -1
```

## 8. split() Method

The split() method splits a string into an array of substrings using a specified delimiter, which can be a substring or regular expression.

**Syntax:** string.split(separator)

**Example 1: Split by Comma**

```
let str = "apple,banana,orange";  
console.log(str.split(",")); // Output: [ 'apple', 'banana', 'orange' ]
```

**Example 2: Split by Space**

```
let str = "JavaScript is fun";  
console.log(str.split(" ")); // Output: [ 'JavaScript', 'is', 'fun' ]
```

**Example 3: Split by a Regular Expression**

```
let str = "1. First 2. Second 3. Third";  
console.log(str.split(/\d\./)); // Output: [ '', ' First ', ' Second ', ' Third' ]
```

### ❖ When to Use Regular Expressions?

- Validating input (e.g., email, phone numbers).
- Searching and replacing text.
- Extracting specific data (e.g., numbers, words).
- Splitting strings by patterns.

### ❖ Special Characters:

- .: Matches any character (except newline).
- ^: Matches the beginning of a string.
- \$: Matches the end of a string.
- \d: Matches any digit (0-9).
- \w: Matches word characters (letters, digits, underscore).
- \s: Matches whitespace.
- +: Matches one or more of the preceding characters.
- \*: Matches zero or more of the preceding characters.
- ?: Makes the preceding character optional.

# JAVASCRIPT VALIDATION

- Form Validation
- Email Validation

## ❖ Example of Form Validation:

### ➤ index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Form validation</title>
</head>
<body>
    <form onsubmit="return fsubmit();" action="home.html">
        UserId: <input id="d1" type="text"><br><br>
        MobileNo: <input id="d2" type="tel"><br><br>
        password: <input id="d3" type="password"><br><br>
        ConfirmPwd: <input id="d4" type="password"><br><br>
        <input type="submit" value="Submit">
    </form>
    <script>
        function fsubmit() {
            const a = document.getElementById("d1").value;
            const b = document.getElementById("d2").value;
            const c = document.getElementById("d3").value;
            const d = document.getElementById("d4").value;
            if (a === "" || b === "" || c === "" || d === "") {
                alert("All fields are mandatory");
                return false;
            }
            else if (b.length<10 || b.length>10) {
                alert("Please Enter 10 digits");
                return false;
            }
            else if (isNaN(b)) {
                alert("Please Enter only digits");
                return false;
            }
            else if (c!==d) {
                alert("Password is not same !");
                return false;
            }
            else { return true;
            }
        }
    </script>
</body>
</html>
```

### home.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>data submit successfully </h1>
</body>
</html>
```

UserId:

MobileNo:

password:

ConfirmPwd:



## Example-2 form validation :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Form Validation Example</title>
</head>
<body>
    <h1>Registration Form</h1>
    <form id="myForm" onsubmit="return fsubmit();" action="home.html">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required /><br><br>
        <span class="error" id="nameError"></span>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required /><br><br>
        <span class="error" id="er"></span>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required /><br><br>
        <button type="button" onclick="togglePassword('password')">Show Password</button><span
class="error" id="pr"></span>
        <label for="cp">Confirm Password:</label>
        <input type="password" id="cp" name="confirmPassword" required /><br>
        <span class="error" id="cpr"></span>
        <button type="submit">Register</button>
    </form>
    <script>
        function fsubmit() {
            const nameInput = document.getElementById('name');
            const emailInput = document.getElementById('email');
            const passwordInput = document.getElementById('password');
            const confirmPasswordInput = document.getElementById('cp');
            const nameError = document.getElementById('nameError');
            const emailError = document.getElementById('er');
            const passwordError = document.getElementById('pr');
            const confirmPasswordError = document.getElementById('cpr');

            let isValid = true;
            nameError.textContent = "";
            emailError.textContent = "";
            passwordError.textContent = "";
            confirmPasswordError.textContent = "";

            // name validation
            const namePattern = /^[A-Za-z\s]+$/;
            if (!namePattern.test(nameInput.value)) {
                nameError.textContent = 'Name can only contain
letters and spaces.';
                isValid = false;
            }
        }
    </script>

```

### home.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <h1>data submit successfully </h1>
</body>
</html>
```

### regular expression ^[A-Za-z\s]+\$

- ^: Matches the beginning of the string.
- [A-Za-z]: Matches any uppercase letter (A to Z) or lowercase letter (a to z).
- \s: Matches any whitespace character (spaces, tabs, newlines, etc.).
- +: Matches one or more occurrences of the preceding character or group. In this case, it means "one or more letters or whitespace characters".
- \$: Matches the end of the string. **So, put together:**
- ^[A-Za-z\s]+\$: Matches a string that starts at the beginning (^), contains one or more (+) letters (A-Z, a-z) or whitespace characters (\s), and ends at the end of the string (\$).



### // email validation

```
const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
```

```
if (!emailPattern.test(emailInput.value)) {
    emailError.textContent = 'Invalid email address.';
    isValid = false;
}
```

### // password validation

```
const passwordPattern = /^(?=.*[A-Z])(?=.*![#$%^&*()_+=\{\};:\\"\\|,.<>\?]).{8,}$/;
```

```
if (!passwordPattern.test(passwordInput.value)) {
    passwordError.textContent = 'Password must:<ul><li>Start with a capital letter</li><li>Contain at least one special symbol</li><li>Be at least 8 characters long</li><li>Not contain spaces</li></ul>';
    isValid = false;
}
```

### // Confirm password

```
if (passwordInput.value !== confirmPasswordInput.value) {
    confirmPasswordError.textContent = 'Passwords do not match.';
    isValid = false;
}
return isValid;
}
```

### // show the password

```
function togglePassword(inputId) {
    const passwordInput = document.getElementById(inputId);
    const type = passwordInput.getAttribute('type') === 'password' ? 'text' : 'password';
    passwordInput.setAttribute('type', type);
}
</script>
</body>
</html>
```

## Registration Form

Name:

Email:

Password:

Confirm Password:

### ❖ Email validation regular expression

`^[^\s@]+@[^\s@]+\.[^\s@]+$`

- ^: Matches the beginning of the string.
- [^\s@]: Matches any character that is *not* a whitespace character (\s) or an @ symbol. The ^ inside the square brackets [] means "not".
- +: Matches one or more occurrences of the preceding character or group.
- @: Matches the literal @ symbol.
- \.: Matches the literal . symbol. The backslash \ is used to escape the dot, as the dot has a special meaning in regular expressions (it matches any character except a newline).
- \$: Matches the end of the string.

### password validation regular expression

`^(?=.*[A-Z])(?=.*![#$%^&*()_+=\{\};:\\"\\|,.<>\?]).{8,}$`

- **At least one uppercase letter:** (?=.\*[A-Z]) (positive lookahead: checks if there's an uppercase letter anywhere in the string without consuming characters).
- **At least one special character:** (?=.\*![#\$%^&\*()\_+=\{\};:\\"\\|,.<>\?]) (positive lookahead: checks for a special character).
- **Minimum length of 8 characters:** .{8,} (matches any character at least 8 times).
- **No spaces:** Spaces are implicitly disallowed because \s (space) is not included in the allowed character sets.
- **Start to End Match:** ^ and \$ ensure the entire string matches the pattern.



# Web APIs

## ❖ What is Web API?:

- API stands for Application Programming Interface.
- A Web API is an application programming interface for the Web.
- A Browser API can extend the functionality of a web browser.
- A Server API can extend the functionality of a web server.

## ❖ Browser APIs:

- All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.
- For example, the Geolocation API can return the coordinates of where the browser is located.

### Example:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Geolocation</h2>
<p>Click the button to get your coordinates.</p>
<button onclick="getLocation()">Try It</button>
<p id="demo"></p>
<script>
const x = document.getElementById("demo");
function getLocation() {
  try {
    navigator.geolocation.getCurrentPosition(showPosition);
  } catch {
    x.innerHTML = err;
  }
}
function showPosition(position) {
  x.innerHTML = "Latitude: " + position.coords.latitude +
    "<br>Longitude: " + position.coords.longitude;
}
</script>
</body>
</html>
```

## ❖ Third Party APIs:

- Third party APIs are not built into your browser.
- To use these APIs, you will have to download the code from the Web.

### Examples:

- YouTube API - Allows you to display videos on a web site.
- Twitter API - Allows you to display Tweets on a web site.
- Facebook API - Allows you to display Facebook info on a web site.

## ❖ JavaScript Validation API:

- JavaScript Validation API, also known as the Constraint Validation API, is a set of built-in browser functions and properties that enable client-side form validation in web applications. It provides a convenient way to validate user input in HTML forms before submitting them to the server. The Validation API is available in modern web browsers and can be accessed using JavaScript.
- ❖ Features and components of the JavaScript Validation API

**1. HTML5 Validation Attributes:** HTML5 introduced several new attributes that can be added to form elements to specify validation rules. These attributes include `required`, `min`, `max`, `pattern`, and more. They are used to define constraints on form fields.

```
<input type="text" id="username" name="username" required minlength="3" maxlength="20">
```

**2. Constraint Validation Methods:** The Validation API provides methods that can be called on form elements to check their validity. The most commonly used methods include:

- `checkValidity()`: Checks if the input field's value meets all defined constraints and returns a Boolean (`true` if valid, `false` if not).
- `setCustomValidity(message)`: Allows you to set a custom validation message that is displayed when the `checkValidity()` method returns `false`.
- `reportValidity()`: Displays the validation message if the input is invalid, or returns `true` if the input is valid.

**3. Validity Properties:** Each form element has a set of validity properties that can be accessed to check its validation state. Common validity properties include:

- `validity.valid`: A Boolean indicating whether the field's value is valid.
- `validity.valueMissing`: A Boolean indicating whether the field is required and its value is empty.
- `validity.tooShort` and `validity.tooLong`: Booleans indicating whether the input length is less than or greater than the specified `minlength` or `maxlength`.

**4. Validation Messages:** When an input field is invalid, the Validation API displays a default validation message. However, you can customize these messages using the `setCustomValidity()` method or by defining the `title` attribute on the input element.

**5. Styling for Invalid Fields:** Invalid form elements can be styled using the `:invalid` CSS pseudo-class. This allows you to apply specific styles to invalid fields, such as changing the border color to red.

```
input:invalid {  
    border-color: red;  
}
```

**6. Form Submission Prevention:** When using Validation API, if a form is invalid (e.g., required fields are empty), it prevents the form from being submitted, giving users a chance to correct their input.

**7. Event Handling:** You can listen for form-related events, such as `submit`, and use JavaScript to check form validity and prevent submission if necessary. Additionally, you can listen for events like `input` and `change` on form fields to provide real-time feedback to users as they fill out the form.

```
const form = document.getElementById('myForm');  
form.addEventListener('submit', function(event) {  
    if (!form.checkValidity()) {  
        event.preventDefault(); // Prevent form submission  
        // Display error messages or take other actions  
    }  
});
```



# **JS AJAX**

- AJAX (Asynchronous JavaScript and XML)\*\* is a set of web development techniques used to create asynchronous web applications. It allows you to exchange data with a web server without having to reload the entire web page. AJAX leverages a combination of technologies, including JavaScript, XML (though often replaced with JSON), and the XMLHttpRequest object (or newer alternatives like Fetch API).

## **1. XMLHttpRequest Object:**

- The `XMLHttpRequest` object is a key component of AJAX. It provides a way to communicate with a web server from a web page using JavaScript.
- It can send HTTP requests (GET, POST, PUT, DELETE, etc.) to a server and handle the server's response asynchronously.

## **2. Asynchronous Requests:**

- AJAX allows you to make asynchronous requests, meaning that the web page doesn't need to wait for the response before continuing to process other tasks.
- This asynchronous behavior enhances the user experience by preventing the page from freezing during data retrieval.

## **3. Data Formats:**

- Originally, AJAX used XML for data interchange, and the acronym itself includes "XML." However, JSON (JavaScript Object Notation) has become the preferred format due to its simplicity, ease of use, and compatibility with JavaScript.
- You can also use other formats like plain text or HTML, depending on your requirements.

## **4. AJAX Workflow:**

- An AJAX request typically involves the following steps:
1. Create an XMLHttpRequest object.
  2. Define a callback function to handle the server's response.
  3. Open a connection to the server, specifying the HTTP method and URL.
  4. Send the request to the server.
  5. Receive and process the server's response in the callback function.

## **5. Callback Functions:**

- Callback functions are essential in AJAX to handle server responses.
- Common callback functions include `onreadystatechange`, which monitors the state of the request, and functions like `onload`, `onerror`, and `ontimeout`, which handle specific aspects of the response.

## **6. Same-Origin Policy:**

- AJAX requests are subject to the same-origin policy, which means that web pages can only make requests to the same domain from which they originated. This policy is in place for security reasons.
- To make requests to different domains, techniques like JSONP (JSON with Padding) or Cross-Origin Resource Sharing (CORS) are used.

## **7. Libraries and Frameworks:**

- While you can work with AJAX directly using the XMLHttpRequest object, many libraries and frameworks simplify AJAX interactions. jQuery's AJAX functions, Axios, and the Fetch API are examples of tools that make it easier to work with AJAX.



#### **8. Use Cases:**

- AJAX is commonly used for various purposes, including:
- Loading dynamic content without refreshing the entire page (e.g., single-page applications).
- Implementing auto-suggest search boxes.
- Submitting form data in the background to provide instant feedback.
- Fetching data from APIs to populate web pages with real-time information.
- Implementing chat applications and social media features.

#### **9. Benefits:**

- AJAX enhances user experience by making web pages more interactive and responsive.
- It reduces server load by fetching only the necessary data, rather than reloading entire pages.
- It allows for the creation of web applications that behave more like desktop applications.

#### **10. Considerations:**

- When implementing AJAX, consider accessibility, error handling, and user feedback to ensure a robust and user-friendly experience.
- Always validate and sanitize data received from the server to prevent security vulnerabilities like cross-site scripting (XSS).

**Note:** summary, AJAX is a powerful technology that enables asynchronous communication between web pages and servers, enhancing the interactivity and responsiveness of web applications. It has become an essential tool in modern web development, allowing developers to create dynamic and user-friendly web experiences.



## **JS JSON**

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. JSON is often used to transmit data between a server and a web application or between different parts of an application. It is a text-based format that resembles JavaScript object literal syntax.

❖ **JSON Syntax:**

- JSON consists of key-value pairs, where keys are strings enclosed in double quotes, followed by a colon, and then values. Values can be strings, numbers, objects, arrays, `true`, `false`, or `null`.
- JSON objects are enclosed in curly braces '{}', and JSON arrays are enclosed in square brackets '[]'

**Example JSON Data:**

```
{  
  "name": "sangamkumar",  
  "age": 30,  
  "city": "patan",  
  "isStudent": false,  
  "hobbies": ["reading", "swimming"]  
}
```

❖ **JavaScript and JSON:**

- JavaScript provides methods for working with JSON data:
- `JSON.stringify(obj)` : Converts a JavaScript object into a JSON string.
- `JSON.parse(json)` : Parses a JSON string and returns a JavaScript object.

❖ **JSON.stringify() Example:**

```
const person = {  
  name: "John Doe",  
  age: 30,  
  city: "New York",  
  isStudent: false,  

```

❖ **JSON.parse() Example:**

```
const jsonPerson =  
  '{"name": "Sangam", "age": 30, "city": "patan", "isStudent": false, "hobbies": ["reading", "swimmin  
g"]};  
const person = JSON.parse(jsonPerson);  
console.log(person);
```

**Complete Example:**

➤ **Raj.html**

```
<!DOCTYPE html>  
<html>  
<head>  
<title>JSON Example</title>  
</head>  
<body>
```



```
<h1>JSON Example</h1>
<p id="json-output"></p>
<script>
    // JavaScript object
    const person = {
        name: "Sangam kumar",
        age: 30,
        city: "patna",
        isStudent: false,
        hobbies: ["reading", "swimming"]
    };
    // Convert JavaScript object to JSON string
    const jsonPerson = JSON.stringify(person);
    document.getElementById('json-output').innerText = 'JSON Data:\n' + jsonPerson;
    // Parse JSON string back to JavaScript object
    const parsedPerson = JSON.parse(jsonPerson);
    console.log(parsedPerson);
</script>
</body>
</html>
```



# **JS vs jQuery**

➤ JavaScript (JS) and jQuery are both tools used for web development, but they serve different purposes and have their own strengths and weaknesses. Let's compare JavaScript and jQuery:

## ❖ **JavaScript:**

### 1. Core Web Language:

- JavaScript is a fundamental web programming language. It is supported by all modern web browsers, and it's the foundation for building interactive web applications.

### 2. DOM Manipulation:

- JavaScript allows you to directly interact with the Document Object Model (DOM), which represents the structure of a web page. You can access and manipulate HTML elements, attributes, and styles with pure JavaScript.

### 3. Full Control:

- JavaScript provides complete control over every aspect of a web page, making it highly versatile for complex web applications

### 4. Community and Ecosystem:

- JavaScript has a vast and active community, along with a wide range of libraries and frameworks (e.g., React, Angular, Vue.js) that can be used to simplify and enhance web development.

### 5. Learning Curve:

- JavaScript can have a steeper learning curve, especially for beginners, due to its complex features and the need to manage browser compatibility.

### 6. Performance:

- When used efficiently, JavaScript can offer excellent performance as it allows you to optimize code for specific use cases.

## ❖ **jQuery:**

### 1. JavaScript Library:

- jQuery is a popular JavaScript library that simplifies many common tasks in web development. It is built on top of JavaScript and provides a higher-level API for working with the DOM.

### 2. DOM Manipulation Simplified:

- jQuery abstracts the complexities of DOM manipulation, allowing developers to perform common operations with less code. For example, selecting and manipulating elements is more concise.

### 3. Cross-Browser Compatibility:

- jQuery handles many cross-browser compatibility issues under the hood, making it easier to write code that works consistently across different browsers.

### 4. Extensive Plugin Ecosystem:

- jQuery has a vast ecosystem of plugins that extend its functionality, making it easy to add features like sliders, carousels, and UI components to websites.

### 5. Rapid Development:

- For quick prototypes or smaller projects, jQuery can significantly speed up development, reducing the amount of code that needs to be written.

### 6. Learning Curve:

jQuery has a lower learning curve compared to pure JavaScript, making it accessible to beginners and those who want to quickly enhance their websites.



## **JS Graphics**

- it allows you to create and manipulate graphical elements on web pages, enabling the development of interactive and visually appealing web applications. Graphics in JavaScript are typically created using the HTML5 <canvas> element or SVG (Scalable Vector Graphics). Below, I'll provide an overview of both approaches, along with examples using HTML and CSS:

### **1. HTML5 <canvas> for Raster Graphics:**

- The <canvas> element provides a raster graphics environment where you can draw and manipulate pixel-based images. Here are some key points:
- Drawing Context: To work with <canvas>, you obtain a drawing context using getContext('2d'). The 2D context (ctx) provides methods and properties for drawing shapes, text, and images.
- Shapes: You can draw various shapes, including rectangles, circles, lines, and paths, on the canvas. These shapes can be filled with colors or patterns.
- Images: You can load and display images on the canvas using the drawImage method, allowing you to create image galleries, games, and more.
- Text: The canvas allows you to render text with different fonts, sizes, and styles. You can position text wherever you like on the canvas.
- Animation: JavaScript can be used to create animations on <canvas>. By repeatedly redrawing the canvas at specific intervals, you can create interactive animations and games.
- Event Handling: You can handle user interactions such as mouse clicks, mouse movement, and keyboard input to create interactive graphics.

#### **Example:**

```
<!DOCTYPE html>
<html>
<head>
<title>Canvas Graphics Example</title>
<style>
    canvas {
        border: 1px solid #000;
    }
</style>
</head>
<body>
<h1>Canvas Graphics Example</h1>
<canvas id="myCanvas" width="400" height="200"></canvas>
<script> // Get the canvas element and its drawing context
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');// Draw a red rectangle
ctx.fillStyle = 'red';
ctx.fillRect(50, 50, 200, 100);// Draw text
ctx.fillStyle = 'blue';
ctx.font = '24px Arial';
ctx.fillText('Hello, Canvas!', 100, 180);
</script>
</body></html>
```

## 2. SVG (Scalable Vector Graphics) for Vector Graphics:

- SVG is a markup language that describes vector graphics using XML. It offers a powerful way to create graphics that can be scaled without loss of quality. Here are some key points:
- Vector Graphics: SVG is ideal for creating vector graphics, which are based on mathematical formulas that describe shapes. This means that SVG images can be scaled up or down without losing detail or quality.
- Elements: SVG uses XML elements to define graphics. Common SVG elements include `<rect>` for rectangles, `<circle>` for circles, `<path>` for custom shapes, `<text>` for text, and more.
- Attributes: SVG elements are styled using attributes like `fill` (for colors), `stroke` (for outlines), `stroke-width` (for line thickness), and others.
- Interactive: SVG graphics can be made interactive by adding event listeners to SVG elements, allowing you to respond to user actions like clicks and mouseovers.
- Animation: SVG supports animations and transitions using CSS, JavaScript, or declarative animations defined within the SVG itself.
- Accessibility: SVG images are accessible by default, making them a good choice for creating graphics that are compatible with screen readers and assistive technologies.
- Integration: SVG graphics can be embedded directly into HTML documents as inline SVG or included as external files. This makes SVG a flexible choice for creating illustrations, icons, charts, and more.

### Example:

```
<!DOCTYPE html>
<html>
<head>
<title>SVG Graphics Example</title>
</head>
<body>
<h1>SVG Graphics Example</h1>
<svg width="400" height="200">
<rect x="50" y="50" width="200" height="100" fill="red" />
<text x="100" y="180" font-family="Arial" font-size="24" fill="blue">Hello, SVG!</text>
</svg>
</body>
</html>
```



# **EXCEPTION HANDLING IN JS**

- Exception handling in JavaScript is a mechanism that allows developers to gracefully handle errors and unexpected situations in their code.
- JavaScript provides a set of constructs for handling exceptions to prevent script termination and provide useful information about the error.

**Example:**

## **1. Types of Errors:**

- In JavaScript, there are several types of errors, including:
  - **Syntax Errors:** Occur when the code violates the language syntax rules. These errors prevent code execution and are usually caught during script parsing.
  - **Runtime Errors (Exceptions):** Occur during script execution when something goes wrong, such as division by zero, referencing an undefined variable, or trying to access properties of null or undefined objects.
  - **Logical Errors:** Do not generate exceptions but result in incorrect behavior due to flawed logic in code.

## **2. try...catch` Statement:**

- The primary mechanism for handling exceptions in JavaScript is the `try...catch` statement.
- It allows you to wrap potentially error-prone code within a `try` block and specify how to handle exceptions in the `catch` block.

**Example:**

```
try {  
    // Code that may throw an exception  
} catch (error) {  
    // Code to handle the exception  
}
```

## **3. Error Object:**

- When an exception is thrown, JavaScript creates an `Error` object that contains information about the error, including the error message and stack trace.
- You can access this object through the `catch` block's parameter.

**Example:**

```
try {  
    // Code that may throw an exception  
} catch (error) {  
    console.error(error.message);  
}
```

## **4. finally Block:**

- The `finally` block, if present, is executed regardless of whether an exception was thrown or caught.
- It is often used for cleanup tasks like closing files or releasing resources.

**Example:**

```
try {  
    // Code that may throw an exception  
} catch (error) {  
    // Code to handle the exception  
} finally {// Cleanup code  
}
```

### 5. Throwing Custom Errors:

- Developers can throw custom errors using the `throw` statement.
- This allows you to create and propagate custom error messages with specific details.

#### Example:

```
function divide(a, b) {  
    if (b === 0) {  
        throw new Error('Division by zero is not allowed.');//  
    }  
    return a / b;  
}
```

### 6. Built-in Error Types:

- JavaScript provides several built-in error types, such as `Error`, `SyntaxError`, `TypeError`, `ReferenceError`, and more.
- You can catch specific error types to handle them differently.

#### Example:

```
try {  
    // Code that may throw an exception  
} catch (error) {  
    if (error instanceof TypeError) {  
        console.error('Type error occurred.');//  
    } else {  
        console.error('An error occurred:', error.message);  
    }  
}
```

### 7. Asynchronous Exception Handling:

- For asynchronous code, such as `'setTimeout'` or AJAX requests, exceptions do not propagate to the global scope.
- You should handle exceptions within the asynchronous code or use mechanisms like `'Promise'` rejections or `'async/await'` to handle errors.

### 8. Global Error Handling:

- You can set a global error handler using `'window.onerror'` to catch unhandled exceptions and log or display custom error messages.

#### Example:

```
window.onerror = function (message, source, lineno, colno, error) {  
    console.error('An unhandled error occurred:', error.message);  
    return true; // Prevent default browser error handling};
```

### Example-1: Handling a potential error when accessing an object property:

```
function getFirstName(person) {  
    try {  
        return person.name.first; // Might cause an error if 'name' or 'first' is missing  
    } catch (error) {  
        return "Unknown"; // Return a default value  
    }  
}  
  
const person1 = { name: { first: "Sangam" } };  
const person2 = {};// Missing 'name' property  
console.log(getFirstName(person1)); // Output: Sangam  
console.log(getFirstName(person2)); // Output: Unknown
```

**Example-2: Handling potential errors with parseInt:**

```
function convertToNumber(str) {  
    try {  
        const num = parseInt(str); // Might cause NaN if str is not a number  
        if (isNaN(num)) {  
            throw new Error("Not a valid number"); // Throw error if NaN  
        }  
        return num * 2;  
    } catch (error) {  
        return "Invalid input";  
    }  
}  
console.log(convertToNumber("10")); // Output: 20  
console.log(convertToNumber("hello")); // Output: Invalid input
```

**Example-3: Simple division with error handling**

```
function divide(a, b) {  
    try {  
        if (b === 0) {  
            throw new Error("Cannot divide by zero");  
        }  
        return a / b;  
    } catch (error) {  
        return error.message; // Return the error message  
    }  
}  
console.log(divide(10, 2)); // Output: 5  
console.log(divide(10, 0)); // Output: Cannot divide by zero
```

**Example-4:**

```
function divide(a, b) {  
    try {  
        if (typeof a !== 'number' || typeof b !== 'number') {  
            throw new TypeError("Inputs must be numbers.");  
        }  
        if (b === 0) {  
            throw new Error("Cannot divide by zero.");  
        }  
        return a / b;  
    } catch (error) {  
        return error.message; // Return the error message  
    } finally {  
        console.log("Division complete"); // Always run  
    }  
}  
console.log(divide(10, 2)); // Output: 5 \n Division complete  
console.log(divide(10, 0)); // Output: Cannot divide by zero. \n Division complete  
console.log(divide(10, "hello")); // Output: Inputs must be numbers. \n Division complete  
console.log(divide(undefined, 5)) // Output: Inputs must be numbers. \n Division complete
```

### Example-5:- Handling a missing object property:

```
function greetUser(user) {  
    try {  
        const name = user.name;  
        console.log("Hello, " + name);  
    } catch (error) {  
        console.log("Hello, Guest!"); // Default greeting if name is missing  
    }  
}  
  
const user1 = { name: "Sangam" };  
const user2 = {}; // No 'name' property  
  
greetUser(user1); // Output: Hello, Sangam  
greetUser(user2); // Output: Hello, Guest!
```

### Example-6: Handling invalid input to a function

```
function doubleNumber(str) {  
    try {  
        const num = Number(str); // Try to convert to a number  
        if (isNaN(num)) { // Check if the conversion resulted in NaN (Not a Number)  
            throw "Invalid Number" // throw custom error  
        }  
        return num * 2;  
    } catch (error) { // in place of error any valid message can be  
        return "Please enter a valid number.";  
    }  
}  
  
console.log(doubleNumber("5")); // Output: 10  
console.log(doubleNumber("hello")); // Output: Please enter a valid number.
```

### Example-7: Handling division by zero:

```
function divideNumbers(a, b) {  
    try {  
        if (b === 0) {  
            throw "Cannot divide by zero." // Throw an error if b is 0  
        }  
        return a / b;  
    } catch (error) {  
        return error; // Return the error message  
    }  
}  
  
console.log(divideNumbers(10, 2)); // Output: 5  
console.log(divideNumbers(10, 0)); // Output: Cannot divide by zero.
```

- **try block:** Contains the code that *might* cause an error.
- **catch block:** Contains the code that runs *if* an error occurs in the try block. The catch block receives the error object (often named error).
- **throw:** Use the throw statement to create and throw a custom error.

## **LOCAL STORAGE IN JS**

- Local Storage is a web storage solution in JavaScript that allows web applications to store key-value pairs in a client's web browser. It provides a simple way to store and retrieve data persistently on the user's device.

- **overview of how to use Local Storage in JavaScript:**

### **1. Storing Data:**

- You can store data in Local Storage using the localStorage.setItem(key, value) method. Both key and value should be strings.

#### **Example:**

```
localStorage.setItem("username", "raj");
localStorage.setItem("email", "ra393@.com");
```

### **2. Retrieving Data:**

- To retrieve data from Local Storage, use the localStorage.getItem(key) method, where key is the string associated with the data you want to retrieve.

#### **Example:**

```
let a=localStorage.getItem("username")
let b=localStorage.getItem("email")
console.log(a)
console.log(b)
```

### **3. Updating Data:**

- You can update existing data by overwriting it with the setItem method.
- localStorage.setItem('username', 'Updatedsangam');

#### **Example:**

```
localStorage.setItem('username', 'Sangam'); // Save initial data
localStorage.setItem('username', 'Sangam123'); // Update existing data
let a=localStorage.getItem("username")
let b=localStorage.getItem("email")
console.log(a)
console.log(b)
```

### **4. Checking for Item Existence:**

- You can check if a specific item exists in Local Storage using localStorage.getItem(key). If the item is not found, it returns null. Answer will show in **true or false**

- **Example:**

```
let a=localStorage.getItem("email")!==null
let b=localStorage.getItem("username")!==null
console.log(a)
console.log(b)
```

### **5. Removing Data:**

- To remove data, use the localStorage.removeItem(key) method.

- **Example:**

```
localStorage.removeItem("email")
let a=localStorage.getItem("username")
let b=localStorage.getItem("email")
console.log(a)
console.log(b)
```

## 6. Clearing All Data:

- To remove all data stored in Local Storage, use the `localStorage.clear()` method.
- `localStorage.clear();`

## 7. Storage Limitations:

- Local Storage has a storage limit of about 5-10 MB, depending on the browser. Attempting to exceed this limit may result in an error.

## 8. Data Type Limitation:

- Local Storage stores data as strings. If you need to store complex data types like objects or arrays, you should serialize and deserialize them using `JSON.stringify()` and `JSON.parse()`.

### Example-1

```
const user = {
    name: 'sangam',
    email: 'sangam@example.com'
};

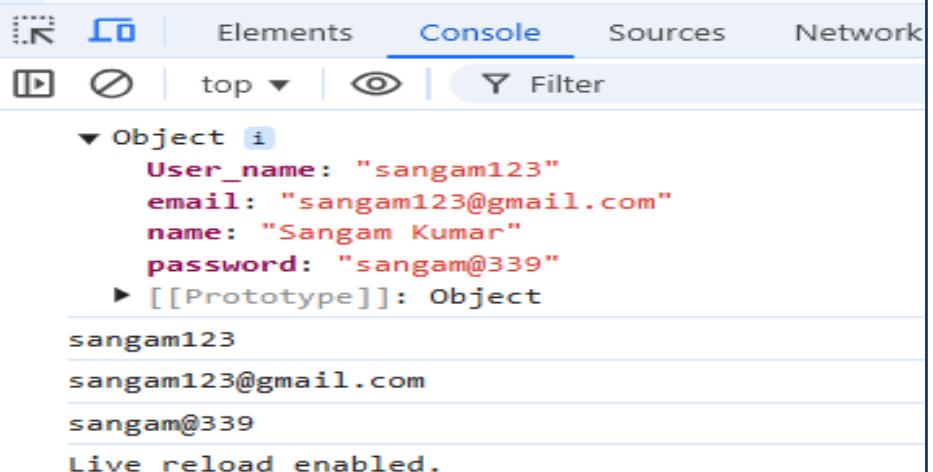
localStorage.setItem('user', JSON.stringify(user));
const retrievedUser = JSON.parse(localStorage.getItem('user'));
```

### Example-2:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">

<title>Local Storage Concept</title>
</head>
<body>
<h1>Local Storage Concept</h1>
<script>
const user = {
    name: 'Sangam Kumar',
    User_name: 'sangam123',
    email: 'sangam123@gmail.com',
    password: 'sangam@339'
};
```

## Local Storage Concept



### // Save user object to localStorage after stringifying it

```
localStorage.setItem('user', JSON.stringify(user));
```

### // Retrieve and parse user object from localStorage

```
const a = JSON.parse(localStorage.getItem('user'));
```

```
console.log(a); // Logs the entire user object
```

```
console.log(a.User_name);
```

```
console.log(a.email);
```

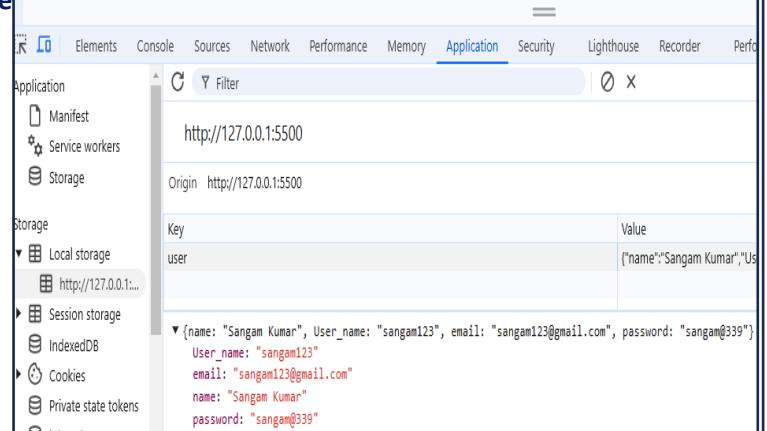
```
console.log(a.password);
```

```
</script>
```

```
</body>
```

```
</html>
```

## Local Storage Concept



**Example-2 :**

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Local Storage Concept</title>
</head>
<body>
<h1>Local Storage Concept</h1>
<script>

// Save initial data to localStorage
localStorage.setItem("username", "raj");
localStorage.setItem("email", "ra393@.com");

// Retrieve and log the data
let a = localStorage.getItem("username");
let b = localStorage.getItem("email");
console.log(a); // Output: raj
console.log(b); // Output: ra393@.com

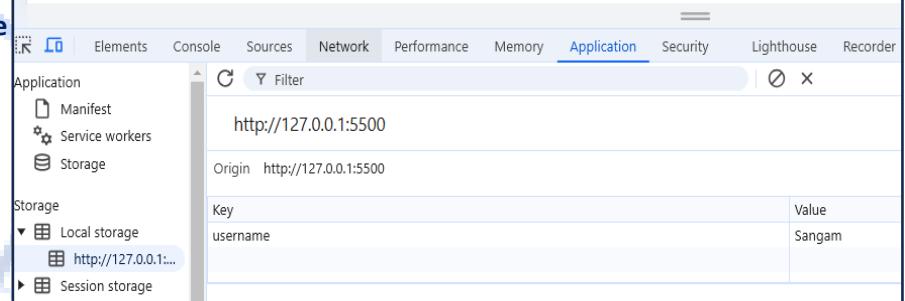
// Update the username
localStorage.setItem("username", "Sangam");
a = localStorage.getItem("username");
b = localStorage.getItem("email");
console.log(a); // Output: Sangam
console.log(b); // Output: ra393@.com

// Remove the email from localStorage
localStorage.removeItem("email");
a = localStorage.getItem("username");
b = localStorage.getItem("email");
console.log(a); // Output: Sangam
console.log(b); // Output: null

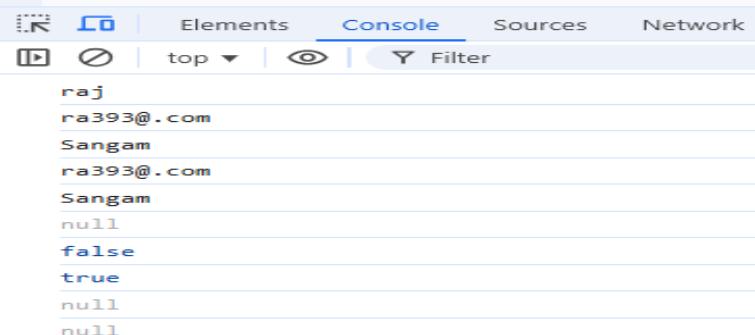
// Check if items exist in localStorage
const emailExists = localStorage.getItem("email") !== null;
const usernameExists = localStorage.getItem("username") !== null;
console.log(emailExists); // Output: false
console.log(usernameExists); // Output: true

// Clear all data from localStorage
localStorage.clear();
a = localStorage.getItem("username");
b = localStorage.getItem("email");
console.log(a); // Output: null
console.log(b); // Output: null
</script>
</body>
</html>
```

### Local Storage Concept



### Local Storage Concept



➤ **index.html**

```
<!DOCTYPE html>
<html>
<head>
<title>Local Storage Example</title>
</head>
<body>

<label for="name">Enter your name:</label>
<input type="text" id="name"><br><br>
<button onclick="saveName()">Save Name</button>
<p id="savedName"></p>
<script>
function saveName() {
    const nameInput = document.getElementById('name');
    const savedName = document.getElementById('savedName');
    const name = nameInput.value;
    if (name.trim() !== '') {
        localStorage.setItem('user_name', name);
        savedName.textContent = `Your name is: ${name}`;
        nameInput.value = "";// Optional: Clear the input field after saving
    } else {
        alert('Please enter a valid name.');
    }
}

// Check if a name is already saved in Local Storage when the page loads
window.onload = function() { // Use window.onload
    const storedName = localStorage.getItem('user_name');
    if (storedName) {
        const savedName = document.getElementById('savedName');
        savedName.textContent = `Your name is: ${storedName}`;
    }
};

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

**Enter your name:**

**Save Name**

**Your name is: Sangam Kumar**



### Example : store user name and password in local storage

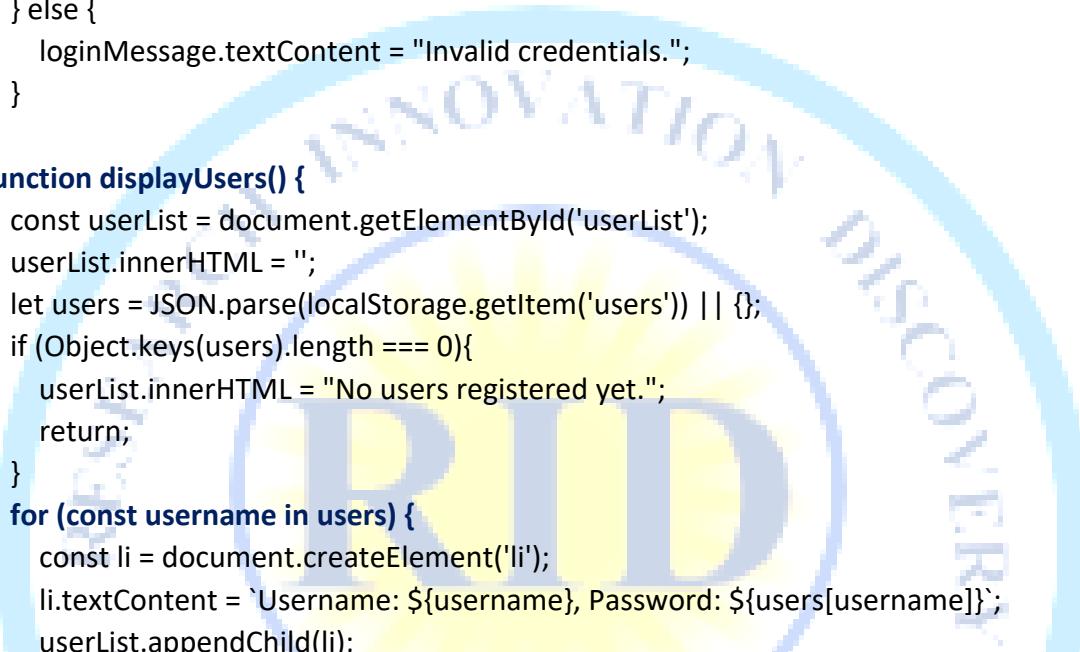
```
<!DOCTYPE html>
<html>
<head>
    <title>User Login/Registration</title>
</head>
<body>
    <h2>Register</h2>
    <input type="text" id="regUsername" placeholder="Username"><br><br>
    <input type="password" id="regPassword" placeholder="Password"><br><br>
    <button onclick="registerUser()">Register</button>
    <p id="regMessage"></p>

    <h2>Login</h2>
    <input type="text" id="loginUsername" placeholder="Username"><br><br>
    <input type="password" id="loginPassword" placeholder="Password"><br><br>
    <button onclick="loginUser()">Login</button>
    <p id="loginMessage"></p>

    <h2>Registered Users</h2>
    <button onclick="displayUsers()">Show Users</button>
    <ul id="userList"></ul>
    <script>
        function registerUser() {
            const username = document.getElementById('regUsername').value.trim();
            const password = document.getElementById('regPassword').value.trim();
            const regMessage = document.getElementById('regMessage');
            if (!username || !password) {
                regMessage.textContent = "Please enter both username and password.";
                return;
            }
            let users = JSON.parse(localStorage.getItem('users')) || {};
            if (users[username]) {
                regMessage.textContent = "Username already exists.";
                return;
            }
            users[username] = password;
            localStorage.setItem('users', JSON.stringify(users));
            regMessage.textContent = "Registration successful!";
            document.getElementById('regUsername').value = "";
            document.getElementById('regPassword').value = "";
            displayUsers();
        }
    </script>

```

```
function loginUser() {
    const username = document.getElementById('loginUsername').value.trim();
    const password = document.getElementById('loginPassword').value.trim();
    const loginMessage = document.getElementById('loginMessage');
    let users = JSON.parse(localStorage.getItem('users')) || {};
    if (users[username] === password) {
        loginMessage.textContent = "Login successful!";
        document.getElementById('loginUsername').value = "";
        document.getElementById('loginPassword').value = "";
    } else {
        loginMessage.textContent = "Invalid credentials.";
    }
}
function displayUsers() {
    const userList = document.getElementById('userList');
    userList.innerHTML = '';
    let users = JSON.parse(localStorage.getItem('users')) || {};
    if (Object.keys(users).length === 0){
        userList.innerHTML = "No users registered yet.";
        return;
    }
    for (const username in users) {
        const li = document.createElement('li');
        li.textContent = `Username: ${username}, Password: ${users[username]}`;
        userList.appendChild(li);
    }
}
</script>
</body>
</html>
```



**Register**

**Register**

**Username already exists.**

**Login**

**Login**

**Registered Users**

**Show Users**

- Username: rajesh, Password: 123
- Username: raj, Password: 123
- Username: sangam kumar, Password: 123456
- Username: raju, Password: 456

## Local Storage in Node.js

- localStorage is a Web API that is only available in browsers, not in Node.js. You are trying to use localStorage in a Node.js environment, where it is not natively supported.

### How to Fix This

- If you are working in Node.js and want to use something similar to localStorage, you have a few options:

#### 1. Use a localStorage Polyfill

- You can use a package like [node-localstorage](#) to mimic localStorage functionality in Node.js.

#### Steps:

- Install the package:

```
npm install node-localstorage
```

- Update your code:

```
const { LocalStorage } = require('node-localstorage');
const localStorage = new LocalStorage('./scratch');

localStorage.setItem("username", "raj");
localStorage.setItem("email", "ra393@.com");

console.log(localStorage.getItem("username")); // Output: raj
console.log(localStorage.getItem("email")); // Output: ra393@.com
```

#### 2. Use fs Module for File Storage

- If you don't want to use external libraries, you can use the fs module to write and read data to/from a file.

#### Example:

```
const fs = require('fs');
const data = {
  username: "raj",
  email: "ra393@.com"
};

// Save to file
fs.writeFileSync('storage.json', JSON.stringify(data));
// Read from file
const storedData = JSON.parse(fs.readFileSync('storage.json', 'utf8'));
console.log(storedData.username); // Output: raj
console.log(storedData.email); // Output: ra393@.com
```

## Example for accordian

```
<!DOCTYPE html>
<html>
<head>
<title>Accordion Example</title>
</head>
<body>
<h2>Accordion Example</h2>
<button class="accordion">Section 1</button>
<div class="panel">
<p>write the data here...</p>
</div>
<button class="accordion">Section 2</button>
<div class="panel">
<p>write the data here .</p>
</div>

<button class="accordion">Section 3</button>
<div class="panel">
<p>write the data here .</p>
</div>
<script>
const acc = document.getElementsByClassName("accordion");
let i;
for (i = 0; i < acc.length; i++) {
  acc[i].addEventListener("click", function() {
    this.classList.toggle("active");
    const panel = this.nextElementSibling;
    panel.classList.toggle("show")
    // if (panel.style.display === "block") {
    //   panel.style.display = "none";
    // } else {
    //   panel.style.display = "block";
    // }
  });
}
</script>
</body>
</html>
```

```
<style>
.accordion {
  background-color: #eee;
  color: #444;
  cursor: pointer;
  padding: 18px;
  width: 100%;
  border: none;
  text-align: left;
  outline: none;
  font-size: 15px;
  transition: 0.4s;
}

.active, .accordion:hover {
  background-color: #ccc;
}

.panel {
  padding: 0 18px;
  display: none;
  background-color: white;
  overflow: hidden;
}

.panel.show{
  display: block;
}
</style>
```

### Accordion Example

Section 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Section 2

Section 3



## Example for Carousel

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>Image Carousel</title>
```

```
<style>
* {box-sizing: border-box}
body {font-family: Verdana, sans-serif;
margin:0}
.mySlides {display: none}
img {vertical-align: middle;}
/* Slideshow container */
.slideshow-container {
max-width: 1000px;
position: relative;
margin: auto;
/* Next & previous buttons */
.prev, .next {
cursor: pointer;
position: absolute;
top: 50%;
width: auto;
padding: 16px;
margin-top: -22px;
color: white;
font-weight: bold;
transition: 0.6s ease;
border-radius: 0 3px 3px 0;
user-select: none;
}
.next {
right: 0;
border-radius: 3px 0 0 3px;
/* On hover, add a black background
color with a little bit see-through */
.prev:hover, .next:hover {
background-color: rgba(0,0,0,0.8); }
}
```

```
/* Caption text */
.text {
color: #f2f2f2;
padding: 8px 12px;
position: absolute;
bottom: 8px;
width: 100%;
text-align: center;
}/* The dots/circles */
.dot {
cursor: pointer;
height: 15px;
width: 15px;
margin: 0 2px;
background-color: #bbb;
border-radius: 50%;
display: inline-block;
transition: background-color 0.6s ease;
}
.active, .dot:hover {
background-color: #717171;
}/* Fading animation */
.fade {
animation-name: fade;
animation-duration: 1.5s;
}
@keyframes fade {
from {opacity: .4}
to {opacity: 1}
}
/* On smaller screens, decrease text size */
@media only screen and (max-width: 300px) {
.prev, .next,.text {font-size: 11px}
}</style>
```

```
</head><body>
```

```
<div class="slideshow-container">
```

```
<div class="mySlides fade">
```

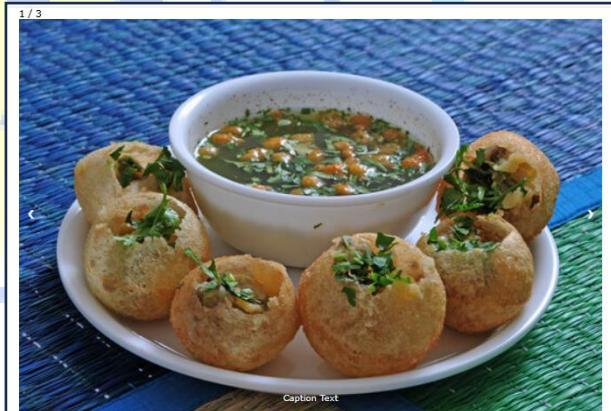
```
<div class="numbertext">1 / 3</div>
```

```

<div class="text">Caption Text</div>
</div>
<div class="mySlides fade">
<div class="numbertext">2 / 3</div>

<div class="text">Caption Two</div>
</div>
<div class="mySlides fade">
<div class="numbertext">3 / 3</div>

<div class="text">Caption Three</div>
</div>
<a class="prev" onclick="plusSlides(-1)">=</a>
<a class="next" onclick="plusSlides(1)">=</a>
</div><br>
<div style="text-align:center">
<span class="dot" onclick="currentSlide(1)"></span>
<span class="dot" onclick="currentSlide(2)"></span>
<span class="dot" onclick="currentSlide(3)"></span> </div>
<script>
let slideIndex = 1;
showSlides(slideIndex);
function plusSlides(n) {
  showSlides(slideIndex += n);
}
function currentSlide(n) {
  showSlides(slideIndex = n);
}
function showSlides(n) {
  let i;
  let slides = document.getElementsByClassName("mySlides");
  let dots = document.getElementsByClassName("dot");
  if (n > slides.length) {slideIndex = 1}
  if (n < 1) {slideIndex = slides.length}
  for (i = 0; i < slides.length; i++) {
    slides[i].style.display = "none";
  }
  for (i = 0; i < dots.length; i++) {
    dots[i].className = dots[i].className.replace(" active", "");
  }
  slides[slideIndex-1].style.display = "block";
  dots[slideIndex-1].className += " active";
}
</script> </body></html>
```



### Example for decrease the time duration

```
<!DOCTYPE html>
<html>
<head>
<title>Countdown Timer</title>
<style>
body {
    font-family: sans-serif;
    text-align: center;
}
#timer {
    font-size: 2em;
}
</style>
</head>
<body>
<h1>Countdown Timer</h1>
<div id="timer">00:00:00</div>
<script>
function startTimer(durationInSeconds) {
    let timerDisplay = document.getElementById('timer');
    let timer = durationInSeconds;
    let hours, minutes, seconds;
    const intervalId = setInterval(function () {
        hours = parseInt(timer / 3600, 10);
        minutes = parseInt((timer % 3600) / 60, 10);
        seconds = parseInt(timer % 60, 10);
        hours = hours < 10 ? "0" + hours : hours;
        minutes = minutes < 10 ? "0" + minutes : minutes;
        seconds = seconds < 10 ? "0" + seconds : seconds;
        timerDisplay.textContent = hours + ":" + minutes + ":" + seconds;
        if (--timer < 0) {
            clearInterval(intervalId);
            timerDisplay.textContent = "Time's up!";
        }
    }, 1000); // Update every 1 second
} // Example usage: Start a 1-hour countdown (3600 seconds)
startTimer(3600);
// You can also start with other durations:
// startTimer(60); // 1 minute
// startTimer(300); // 5 minutes
// startTimer(1800); // 30 minutes
</script></body></html>
```

**Countdown Timer**

00:58:07



## **CONNECT A JAVASCRIPT APPLICATION WITH A MYSQL DATABASE**

- to connect a JavaScript application with a MySQL database, you typically use **Node.js** as the backend runtime environment and the **mysql** or **mysql2** package to handle database operations. Below are step-by-step instructions with code:

### **Step 1: Setup Node.js**

1. Install Node.js from <https://nodejs.org/> if not already installed.

2. Verify installation:

node -v

npm -v

### **Step 2: Initialize a Node.js Project**

1. Create a new project folder and initialize it:

mkdir js-mysql-connection

cd js-mysql-connection

npm init -y

### **Step 3: Install Required Packages**

1. Install the mysql or mysql2 package to interact with MySQL:

npm install mysql

Or, to use mysql2 (recommended for better performance):

bash

CopyEdit

npm install mysql2

### **Step 4: Create a MySQL Database**

1. Set up a MySQL database with a table (if not already created):

CREATE DATABASE mydb;

USE mydb;

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

email VARCHAR(255) NOT NULL

);

### **Step 5: Write the JavaScript Code**

1. Create a file called index.js:

touch index.js

2. Add the following code to connect and interact with the MySQL database:

// Import the mysql module

const mysql = require('mysql'); // Use 'mysql2' if using mysql2 package

```
// Create a connection to the database
const connection = mysql.createConnection({
  host: 'localhost',      // Database host (e.g., localhost or 127.0.0.1)
  user: 'root',           // Your MySQL username
  password: "",           // Your MySQL password
  database: 'mydb'        // Name of the database
});
// Connect to the database
connection.connect((err) => {
  if (err) {
    console.error('Error connecting to the database:', err);
    return;
  }
  console.log('Connected to the MySQL database');
});
// Insert a new user into the 'users' table
const addUser = `INSERT INTO users (name, email) VALUES ('John Doe',
'john@example.com')`;
connection.query(addUser, (err, results) => {
  if (err) {
    console.error('Error inserting data:', err);
    return;
  }
  console.log('User added:', results.insertId);
});

// Retrieve all users
const getUsers = `SELECT * FROM users`;
connection.query(getUsers, (err, results) => {
  if (err) {
    console.error('Error fetching data:', err);
    return;
  }
  console.log('Users:', results);
});

// Close the database connection
connection.end((err) => {
  if (err) {
    console.error('Error closing the connection:', err);
    return;
  }
  console.log('Database connection closed');
});
```



### Step 6: Run the Application

1. Start the Node.js application:  
node index.js
2. Output Example:  
Connected to the MySQL database  
User added: 1  
Users: [ { id: 1, name: 'John Doe', email: 'john@example.com' } ]  
Database connection closed

### Step 7: Troubleshooting

1. **Error: MySQL not running**  
Ensure MySQL is running locally or remotely:  
sudo service mysql start
2. **Access Denied Errors**  
Double-check your MySQL credentials (user and password).
3. **Firewall Issues**  
If using a remote database, ensure the MySQL port (default is 3306) is open.

### Additional Notes

- For **production**, consider using environment variables for sensitive data like database credentials. Use the dotenv package:  
npm install dotenv

Example .env file:

```
makefile
CopyEdit
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=
DB_NAME=mydb
```

Modify your code to use environment variables:

```
require('dotenv').config();
const connection = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME
});
```

## CONNECT A JAVASCRIPT APPLICATION WITH A MONGODB DATABASE

To connect a JavaScript application with a MongoDB database, you can use the **mongodb** package. Here's the step-by-step solution with the necessary code:

### Step 1: Install Node.js

1. Download and install **Node.js** from <https://nodejs.org/>.
2. Verify installation:

```
node -v
```

```
npm -v
```

### Step 2: Install MongoDB

1. Install MongoDB on your machine if it's not already installed. Follow the instructions from [MongoDB Installation Guide](#).
2. Start the MongoDB server:  

```
mongod
```

Alternatively, you can use **MongoDB Atlas**, a cloud-hosted MongoDB service, for free. Sign up at <https://www.mongodb.com/atlas>.

### Step 3: Create a New Node.js Project

1. Create a new project folder:  

```
mkdir js-mongodb-connection
```

```
cd js-mongodb-connection
```

```
npm init -y
```

### Step 4: Install Required Packages

1. Install the **mongodb** package:  

```
npm install mongodb
```

### Step 5: Write the Code

1. Create a file named `index.js`:  

```
touch index.js
```
2. Add the following code to connect to the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations:

```
// Import MongoClient from the 'mongodb' package
```

```
const { MongoClient } = require('mongodb');
```

```
// MongoDB connection URL
```

```
const url = 'mongodb://127.0.0.1:27017'; // Use your MongoDB Atlas URI if using Atlas
```

```
const client = new MongoClient(url);
```

```
// Database and Collection
```

```
const dbName = 'mydatabase'; // Change the database name as needed
```

```
const collectionName = 'users';
```



```
// Main function to handle MongoDB operations
async function main() {
  try {
    // Connect to the MongoDB server
    await client.connect();
    console.log('Connected to MongoDB');

    // Select the database and collection
    const db = client.db(dbName);
    const collection = db.collection(collectionName);

    // CREATE: Insert a new document
    const newUser = { name: 'Sangam kuamr', email: 'sangam@example.com', age: 30 };
    const insertResult = await collection.insertOne(newUser);
    console.log('Inserted Document:', insertResult.insertedId);

    // READ: Find all documents
    const users = await collection.find().toArray();
    console.log('All Users:', users);

    // UPDATE: Update a document
    const updateResult = await collection.updateOne(
      { name: 'Sangam kumar' }, // Filter
      { $set: { age: 31 } } // Update
    );
    console.log('Updated Document Count:', updateResult.modifiedCount);

    // DELETE: Delete a document
    const deleteResult = await collection.deleteOne({ name: 'John Doe' });
    console.log('Deleted Document Count:', deleteResult.deletedCount);
  } catch (err) {
    console.error('An error occurred:', err);
  } finally {
    // Close the connection
    await client.close();
    console.log('MongoDB connection closed');
  }
}

// Run the main function
main().catch(console.error);
```

#### Step 6: Run the Application

1. Run the script:

```
node index.js
```

2. Output Example:

Connected to MongoDB

Inserted Document: 61d72e8b98347f7c38c9b9b5

All Users: [ { \_id: ObjectId("61d72e8b98347f7c38c9b9b5"), name: 'John Doe', email: 'john@example.com', age: 30 } ]

Updated Document Count: 1

Deleted Document Count: 1

MongoDB connection closed

#### Step 7: Using MongoDB Atlas (Optional)

1. Go to [MongoDB Atlas](#).
2. Create a free cluster and get your connection string.
3. Replace the url variable in the above code with your Atlas connection string.

Example:

```
const url =
```

```
'mongodb+srv://<username>:<password>@cluster0.mongodb.net/?retryWrites=true&w=majority';
```

#### Step 8: Troubleshooting

1. **MongoDB Server Not Running**

- Start the MongoDB server:  
`mongod`

2. **Authentication Errors**

- Ensure the correct username/password for MongoDB Atlas.
- Whitelist your IP address in Atlas settings.

3. **Deprecation Warnings**

- Use the `{ useUnifiedTopology: true }` option when creating the client if you encounter warnings:

```
const client = new MongoClient(url, { useUnifiedTopology: true });
```

---

#### Next Steps

- Use **Mongoose** for an ORM-like experience:

```
npm install mongoose
```



## **Important question and answer in JavaScript:**

1. What is JavaScript?
  - JavaScript is a versatile and widely used programming language for web development.
2. What are the data types in JavaScript?
  - Primitive types: string, number, boolean, undefined, null, symbol (ES6).
  - Reference types: object, array, function.
3. How do you declare variables in JavaScript?
  - Using var, let, or const.
  - What is the difference between let, const, and var?
  - let and const have block scope, while var has function scope.
  - const cannot be reassigned, let and var can.
4. How do you comment in JavaScript?
  - Using // for single-line comments and /\* \*/ for multi-line comments.
5. What is hoisting in JavaScript?
  - Hoisting is the behavior of moving variable and function declarations to the top of their containing scope during compilation.
6. What is a closure?
  - A closure is a function that retains access to variables from its parent scope even after the parent function has finished executing.
7. How do you define a function in JavaScript?
  - Using the function keyword or as arrow functions (ES6).
8. What is callback hell?
  - Callback hell (or Pyramid of Doom) is a situation where multiple nested callbacks make code hard to read and maintain.
9. What is an anonymous function?
  - An anonymous function is a function without a name.
10. What is the this keyword in JavaScript?
  - this refers to the current object in a function or method.
11. Explain event delegation.
  - Event delegation is a technique where a single event handler is used for multiple elements by using event bubbling.
12. How do you create an object in JavaScript?
  - Using object literals, constructor functions, or class syntax (ES6).
13. What is a prototype in JavaScript?
  - A prototype is an object that provides shared properties and methods for other objects.
14. What is the difference between null and undefined?
  - null is an intentional absence of any object value, while undefined means a variable has been declared but has not been assigned a value.
15. How do you check if a variable is undefined?
  - Using typeof or by comparing to undefined.
16. What is the purpose of the NaN value?
  - NaN represents "Not-a-Number" and is a value returned when a mathematical operation cannot produce a valid result.
17. What is the difference between == and ===?
  - == checks for equality after type coercion, while === checks for strict equality (value and type).

19. How do you convert a string to a number in JavaScript?
  - Using parseInt() or parseFloat().
20. What is the NaN value used for?
  - NaN is used to represent the result of an invalid mathematical operation.
21. What is an IIFE?
  - An IIFE (Immediately Invoked Function Expression) is a function that is defined and executed immediately.
22. What is a promise in JavaScript?
  - A promise is an object representing the eventual completion or failure of an asynchronous operation.
23. How do you handle promises in JavaScript?
  - Using .then() for success and .catch() for errors.
24. What is asynchronous programming in JavaScript?
  - Asynchronous programming allows non-blocking execution of code, typically used for tasks like network requests and timers.
25. Explain the event loop in JavaScript.
  - The event loop is a mechanism that handles asynchronous operations, ensuring that the main thread remains responsive.
26. What is the difference between null and undefined?
  - null is explicitly assigned to indicate the absence of a value, while undefined indicates that a variable has been declared but not assigned a value.
27. What is the purpose of the typeof operator?
  - typeof is used to determine the data type of a variable or expression.
28. How do you add an element to an array in JavaScript?
  - Using the push() method or the spread operator (ES6).
29. What is JSON and how do you parse it in JavaScript?
  - JSON (JavaScript Object Notation) is a data interchange format.  
Use JSON.parse() to parse a JSON string into a JavaScript object.
30. What is the ternary operator in JavaScript?
  - The ternary operator (condition ? expr1 : expr2) is a shorthand for an if-else statement.
31. How do you prevent the default behavior of an event in JavaScript?
  - Using event.preventDefault().
32. What are the localStorage and sessionStorage objects used for?
  - They provide storage for web applications on the client side.
33. How do you loop through an array in JavaScript?
  - Using for, while, or forEach().
34. What is a JavaScript constructor function?
  - A constructor function is used to create and initialize objects.
35. How do you define and use classes in JavaScript (ES6)?
  - Use the class keyword to define classes and the new keyword to create instances.
36. What are arrow functions in JavaScript (ES6)?
  - Arrow functions are concise function expressions that do not bind their own this value.
37. What is the purpose of the map() function in JavaScript?
  - The map() function is used to create a new array by applying a function to each element of an existing array.
38. How do you add and remove classes from HTML elements in JavaScript?

- Use .classList.add() and .classList.remove() methods.
39. Explain the difference between the global scope and local scope.
  - Global scope is accessible from anywhere in the code, while local scope is limited to a specific function or block.
40. What is a callback function in JavaScript?
  - A callback function is a function passed as an argument to another function and executed at a later time.
41. What is a higher-order function in JavaScript?
  - A higher-order function is a function that takes one or more functions as arguments or returns a function.
42. How do you clone an object in JavaScript?
  - Using the spread operator ({ ...obj }) or Object.assign().
43. What is destructuring in JavaScript?
  - Destructuring allows you to extract values from arrays or objects into variables.
44. How do you check if an array contains a specific element in JavaScript?
  - Using the includes() method or indexOf().
45. What is the purpose of the fetch() API in JavaScript?
  - fetch() is used for making network requests and retrieving data from web APIs.
46. What is the purpose of the async and await keywords (ES6)?
  - async is used to define asynchronous functions, and await is used to pause execution until a Promise is resolved or rejected.
47. How do you create and use modules in JavaScript (ES6)?
  - Use import and export statements to define and use modules.
48. What is the difference between null and undefined?
  - null is explicitly assigned to indicate the absence of a value, while undefined indicates that a variable has been declared but not assigned a value.
49. What is the purpose of the try...catch statement in JavaScript?
  - try...catch is used for error handling to catch and handle exceptions.
50. How do you use the localStorage and sessionStorage objects to store data on the client side?
  - Use localStorage.setItem() and localStorage.getItem() (and similarly for sessionStorage) to store and retrieve data.
51. How do you remove an item from an array in JavaScript?
  - Using the splice() method or array methods like filter().
52. What is the difference between let and const?
  - let allows reassignment, while const does not.
53. What is the purpose of the reduce() function in JavaScript?
  - reduce() is used to reduce an array to a single value by applying a function to each element.
54. What is the difference between null and undefined?
  - null is explicitly assigned to indicate the absence of a value, while undefined indicates that a variable has been declared but not assigned a value.
55. How do you create and use JavaScript promises?
  - Promises are created using the Promise constructor and used with .then() and .catch().
56. What is the purpose of the event.preventDefault() method?
  - It prevents the default behavior of an event, such as form submission or link navigation.
57. How do you check if an object has a specific property in JavaScript?
  - Using the hasOwnProperty() method or the in operator.
58. What is the purpose of the Set object in JavaScript (ES6)?

- Set is used to store unique values and provides methods for set operations.
59. How do you convert a string to uppercase or lowercase in JavaScript?
  - Using toUpperCase() and toLowerCase() string methods.
60. What is the purpose of the setTimeout() function in JavaScript?
  - setTimeout() is used to schedule a function to run after a specified delay.
61. How do you create a random number in JavaScript?
  - Using Math.random() and mathematical operations.
62. What is the purpose of the Array.isArray() method?
  - It checks if an object is an array.
63. How do you reverse an array in JavaScript?
  - Using the reverse() method.
64. What is the difference between localStorage and sessionStorage?
  - localStorage persists data until explicitly removed, while sessionStorage data is cleared when the session ends.
65. How do you create and use a JavaScript class (ES6)?
  - Use the class keyword to define a class and the new keyword to create instances.
66. What is the purpose of the splice() method in JavaScript?
  - splice() is used to modify an array by adding or removing elements.
67. How do you check if a value is a number in JavaScript?
  - Using typeof or the isNaN() function.
68. What is the purpose of the pop() and push() methods in JavaScript?
  - pop() removes and returns the last element of an array, while push() adds one or more elements to the end.
69. How do you compare two objects for equality in JavaScript?
  - You need to manually compare their properties and values.
70. What is the purpose of the filter() method in JavaScript?  
filter() is used to create a new array with elements that meet a specified condition.

# What is RID Organization (RID संस्था क्या है)?

- **RID Organization** यानि **Research, Innovation and Discovery Organization** एक संस्था हैं जो TWF (TWKSAA WELFARE FOUNDATION) NGO द्वारा RUN किया जाता है | जिसका मुख्य उद्देश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही इस **RID Organization** की स्थपना 30.09.2023 किया गया है | जो TWF द्वारा संचालित किया जाता है |
- TWF (TWKSAA WELFARE FOUNDATION) NGO की स्थपना 26-10-2020 में बिहार की पावन धरती सासाराम में Er. RAJESH PRASAD एवं Er. SUNIL KUMAR द्वारा किया गया था जो की भारत सरकार द्वारा मान्यता प्राप्त संस्था हैं |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियों से मैं आवाहन करता हूँ की आप सभी इस **RID संस्था** से जुड़ें एवं अपने बुधिद्वय, विवेक एवं प्रतिभा से दुनियां को कुछ नई (**RID, PMS & TLR**) की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें।

## MISSION, VISSION & MOTIVE OF “RID ORGANIZATION”

मिशन	हर एक ONE भारत के संग
विजन	TALENT WORLD KA SHRESHTM AB AAYEGA भारत में और भारत का TALENT भारत में
मकसद	NEW (RID, PMS, TLR)

## MOTIVE OF RID ORGANIZATION NEW (RID, PMS, TLR)

### NEW (RID)

R	I	D
Research	Innovation	Discovery

### NEW (TLR)

T	L	R
Technology, Theory, Technique	Law	Rule

### NEW (PMS)

P	M	S
Product, Project, Production	Machine	Service



RID रीड संस्था की मिशन, विजन एवं मकसद को सार्थक हमें बनाना हैं।  
भारत के वर्चस्व को हर कोने में फैलना हैं।  
कर के नया कार्य एक बदलाव समाज में लाना हैं।  
रीड संस्था की कार्य-सिद्धांतों से ही, हमें अपनी पहचान बनाना हैं।

Er. Rajesh Prasad (B.E, M.E)

Founder:

TWF & RID Organization



**RID BHARAT**

JavaScript के इस E-Book में अगर मिलती त्रुटी मिलती है तो कृपया हमें सूचित करें। WhatsApp's No: 9892782728 or Email Id: [ridorg.in@gmail.com](mailto:ridorg.in@gmail.com)

|| धन्यवाद ||



**RID BHARAT**

Page. No:302

Website: [www.ridbharat.com](http://www.ridbharat.com)