# **🔁 What are Loops in JavaScript?**

**Loops** in JavaScript are used to **execute a block of code repeatedly**, either a fixed number of times or until a certain condition is met.

**✅ Why do we use loops?**

* To **avoid repeating code** manually.
* Automate **repetitive tasks** like iterating over data (arrays, objects, etc.).
* Help with **data processing, UI rendering**, etc.

**🔧 Types of Loops in JavaScript**

JavaScript has **5 main types** of loops:

| **Loop Type** | **Best Use** |
| --- | --- |
| for loop | Fixed iterations (e.g., run 10 times) |
| while loop | Run as long as a condition is true |
| do...while loop | Similar to while, but **runs at least once** |
| for...in loop | Iterate over object **properties** |
| for...of loop | Iterate over **iterables** (like arrays, strings, etc.) |

**🧠 1. for loop**

**🔹 Syntax:**

for (initialization; condition; update) {

// code block

}

**🧪 Example:**

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

console.log("Hello " + i);

}

**✅ Use When:**

* You know how many times you need to loop.

**🧠 2. while loop**

**🔹 Syntax:**

while (condition) {

// code block

}

**🧪 Example:**

let i = 1;

while (i <= 5) {

console.log("Counting " + i);

i++;

}

**✅ Use When:**

* You don’t know the exact number of iterations.

**🧠 3. do...while loop**

**🔹 Syntax:**

do {

// code block

} while (condition);

**🧪 Example:**

let i = 1;

do {

console.log("Run at least once: " + i);

i++;

} while (i <= 3);

**✅ Use When:**

* You want the code to **run at least once** even if the condition is false initially.

**🧠 4. for...in loop**

**🔹 Syntax:**

for (let key in object) {

// access object[key]

}

**🧪 Example:**

let user = {name: "Rahul", age: 25};

for (let key in user) {

console.log(key + ": " + user[key]);

}

**✅ Use When:**

* You want to **iterate through properties of an object**.

**🧠 5. for...of loop**

**🔹 Syntax:**

for (let value of iterable) {

// access value

}

**🧪 Example:**

let fruits = ["apple", "banana", "mango"];

for (let fruit of fruits) {

console.log(fruit);

}

**✅ Use When:**

* You want to **loop over array values, strings, maps, sets**, etc.

**📌 Real-World Examples**

**🎯 1. Show all products in a shopping cart:**

let cart = ["shoes", "shirt", "watch"];

for (let item of cart) {

console.log("Product: " + item);

}

**🎯 2. Validate user inputs until correct:**

let input;

do {

input = prompt("Enter your age:");

} while (isNaN(input));

**🎯 3. Loop through API response object:**

let response = {

status: "OK",

code: 200,

data: "Success"

};

for (let key in response) {

console.log(`${key}: ${response[key]}`);

}

**🆚 Difference: JavaScript Loops vs Java Loops**

| **Feature** | **JavaScript** | **Java** |
| --- | --- | --- |
| Dynamic Typing | ✅ Yes | ❌ No (strict types) |
| for...of / for...in | ✅ Yes | ❌ No (Java has for-each) |
| Scope | let, const have block scope | Java has method/class level scope |
| Arrays and Objects | Can easily loop over both | Needs special handling for objects |
| Runtime Environment | Browser/Node.js | JVM |

In JavaScript, loops are more **flexible and concise** due to the dynamic nature of the language.

**🛠️ Rules / Best Practices**

* Always avoid **infinite loops** → make sure conditions will eventually fail.
* Prefer for...of over for...in when working with arrays.
* Avoid modifying the array you're looping over.
* Use break to exit early, continue to skip current iteration.

**🧾 Summary**

| **Loop** | **Use Case** |
| --- | --- |
| for | Known count |
| while | Unknown count, condition-based |
| do...while | At least one-time execution |
| for...in | Object properties |
| for...of | Array/string values |

### **Var, Let and Const**

In JavaScript, let, var, and const are used to declare **variables**, but they behave differently in terms of **scope**, **hoisting**, and **mutability**. Here's a breakdown of **why** and **when** to use each:

**🔹 var**

**✅ When to use:**

* For **legacy codebases** (pre-ES6, before 2015).
* When you need **function-scoped** variables and understand the **hoisting** implications.

**❌ Avoid when:**

* Writing modern JavaScript – let or const is preferred.
* You want block-level scope.

**⚙️ Characteristics:**

| **Feature** | **Behavior** |
| --- | --- |
| Scope | Function-scoped |
| Hoisting | Hoisted to top of function/scope, **initialized as undefined** |
| Redeclarable | ✅ Yes (in the same scope) |
| Reassignable | ✅ Yes |

**Example:**

function demo() {

console.log(a); // undefined (due to hoisting)

var a = 10;

console.log(a); // 10

}

**🔹 let**

**✅ When to use:**

* When you need a **block-scoped** variable.
* When the value **needs to change** later.

**❌ Avoid when:**

* You don’t plan to reassign the variable (use const instead).

**⚙️ Characteristics:**

| **Feature** | **Behavior** |
| --- | --- |
| Scope | Block-scoped ({}) |
| Hoisting | Hoisted but **not initialized** (TDZ error if accessed before declaration) |
| Redeclarable | ❌ No (in the same scope) |
| Reassignable | ✅ Yes |

**Example:**

let count = 1;

count = 2; // valid

if (true) {

let count = 5; // separate block scope

console.log(count); // 5

}

console.log(count); // 2

**🔹 const**

**✅ When to use:**

* When you **do not want to reassign** the variable.
* For constants, functions, arrays, objects that don’t need reassignment.

**❌ Avoid when:**

* You need to reassign the variable later.

**⚙️ Characteristics:**

| **Feature** | **Behavior** |
| --- | --- |
| Scope | Block-scoped |
| Hoisting | Hoisted but **not initialized** (TDZ applies) |
| Redeclarable | ❌ No |
| Reassignable | ❌ No |

🔸 However, for objects/arrays: You **can mutate** the contents.

**Example:**

const PI = 3.14;

// PI = 3.14159; // ❌ Error

const user = { name: "Gaurav" };

user.name = "Sonar"; // ✅ Allowed (mutation)

**🧠 Summary: When to Use What?**

| **Keyword** | **Use When…** |
| --- | --- |
| const | Default choice. Use for **values that don’t change** (or won't be reassigned). |
| let | Use when value **will change later**. Preferred over var. |
| var | Only use in **legacy codebases** or if you **understand function scope and hoisting well**. |

**🚀 Best Practices**

* **Use const by default**.
* Switch to let if you know the variable will be reassigned.
* **Avoid var** unless maintaining old code or explicitly needing function scope.

## ✅ JavaScript let, var, and const Cheat Sheet

| **Feature** | **var** | **let** | **const** |
| --- | --- | --- | --- |
| Scope | Function-scoped | Block-scoped | Block-scoped |
| Redeclaration | ✅ Allowed in same scope | ❌ Not allowed | ❌ Not allowed |
| Reassignment | ✅ Allowed | ✅ Allowed | ❌ Not allowed (but object mutation is allowed) |
| Hoisting | ✅ Hoisted (initialized as undefined) | ✅ Hoisted (in TDZ\*) | ✅ Hoisted (in TDZ\*) |
| Default Choice | ❌ Legacy only | ✅ If value changes | ✅ If value won’t change |
| Temporal Dead Zone | ❌ No | ✅ Yes | ✅ Yes |

🧠 TDZ = Temporal Dead Zone: Accessing the variable before it's declared will throw a ReferenceError.

📝 Quick Examples  
  
// var

function exampleVar() {

console.log(a); // undefined

var a = 10;

}

// let

function exampleLet() {

// console.log(b); // ReferenceError

let b = 20;

}

// const

const PI = 3.14;

// PI = 3.14159; // ❌ Error

const arr = [1, 2, 3];

arr.push(4); // ✅ Mutation allowed

**📚 Multiple Choice Quiz (with Answers)**

**Q1. Which of the following is block-scoped?**

A. var  
B. let  
C. const  
D. Both B and C  
  
**✅ Answer: D**

**Q2. What happens if you access a let variable before declaration?**

A. undefined  
B. null  
C. ReferenceError  
D. It works fine  
  
**✅ Answer: C**

**Q3. Which variable declaration is hoisted and initialized with undefined?**

A. var  
B. let  
C. const  
D. None  
  
**✅ Answer: A**

**Q4. Can a const variable be reassigned?**

A. Yes  
B. No  
C. Only in loops  
D. Only if it's a number  
  
**✅ Answer: B**

**Q5. What is the output of the following?**

{

let a = 10;

}

console.log(a);

A. 10  
B. undefined  
C. ReferenceError  
D. null  
  
**✅ Answer: C**

**Q6. Which is the best default variable declaration in modern JavaScript?**

A. var  
B. let  
C. const  
D. int  
  
**✅ Answer: C**

**Q7. Which of the following allows redeclaration in the same scope?**

A. const  
B. let  
C. var  
D. None of the above

**✅ Answer: C**

**Q8. What is the output?**

const obj = { name: "John" };

obj.name = "Doe";

console.log(obj.name);

A. Error  
B. John  
C. Doe  
D. undefined  
  
**✅ Answer: C**

### **🧠 What is Hoisting in JavaScript?**

**Hoisting** is JavaScript's default behavior of **moving declarations to the top of their scope** (either global or function scope) **before code execution**.

This means that **you can use variables and functions before declaring them**, but **how they behave depends on how they're declared** (var, let, const, or function).

**🔍 Key Points About Hoisting**

| **Declaration Type** | **Is Hoisted?** | **Initialized on Hoisting?** | **Accessible Before Declaration?** |
| --- | --- | --- | --- |
| var | ✅ Yes | ✅ Initialized as undefined | ✅ Yes, but value is undefined |
| let / const | ✅ Yes | ❌ Not initialized | ❌ ReferenceError (TDZ\*) |
| function | ✅ Yes | ✅ Fully hoisted | ✅ Yes |
| arrow function (with let/const) | ✅ Yes (as variable) | ❌ Not initialized | ❌ ReferenceError |

🧠 *TDZ = Temporal Dead Zone – the time between hoisting and actual declaration where access throws an error.*

**📘 Example: var Hoisting**

console.log(a); // undefined

var a = 5;

console.log(a); // 5

Internally, JavaScript does this:

var a;

console.log(a); // undefined

a = 5;

console.log(a); // 5

**📘 Example: let and const Hoisting**

console.log(b); // ❌ ReferenceError

let b = 10;

console.log(c); // ❌ ReferenceError

const c = 20;

Even though let and const are hoisted, they **are not initialized**. So accessing them before declaration gives **ReferenceError** due to the **Temporal Dead Zone (TDZ)**.

**📘 Example: Function Hoisting**

greet(); // ✅ Works

function greet() {

console.log("Hello!");

}

Function declarations are fully hoisted **with their body**, so you can call them before they're defined.

**📘 Example: Arrow Function with let (Not Fully Hoisted)**

sayHi(); // ❌ ReferenceError

let sayHi = () => console.log("Hi!");

This behaves like:

let sayHi; // hoisted, but uninitialized → TDZ

sayHi(); // ReferenceError

sayHi = () => console.log("Hi!");  
  
  
**🧪 Quiz Yourself**

**What is the output?**

console.log(x);

var x = 10;

✅ Output: undefined

**What is the output?**

console.log(y);

let y = 20;

❌ Output: ReferenceError

**What is the output?**

foo();

function foo() {

console.log("I am foo");

}

✅ Output: I am foo

**🧱 Part 1: JavaScript Dialog Boxes**

These are **built-in browser functions** used to interact with users through **popup dialogs**.

**1️] Alert Box**

**📌 Purpose:**

To **display a message** to the user — **informational only** (no user input).

**🧪 Syntax:**

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

**📋 Rules:**

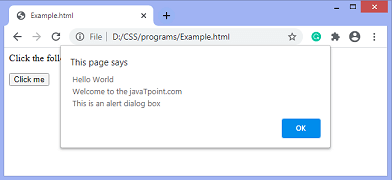
* Only shows an **OK** button.
* Execution **pauses** until the user clicks OK.
* Used mainly for **notifications, warnings**, or **debugging**.

**🎯 Real-World Example:**

if (!isLoggedIn) {

alert("Please login to continue!");

}



**🛠️ When to Use:**

* Displaying messages like:
  + "Form submitted successfully"
  + "Error: Something went wrong"

**2️] Confirm Box**

**📌 Purpose:**

To **ask the user for confirmation** (Yes/No or OK/Cancel).

**🧪 Syntax:**

let result = confirm("Are you sure you want to delete this file?");

**📋 Rules:**

* Returns true if OK is clicked, false if Cancel is clicked.
* Can be used inside if statements to make decisions.

**🎯 Real-World Example:**

let proceed = confirm("Do you really want to logout?");

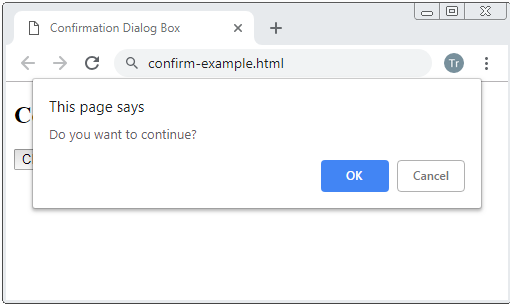
if (proceed) {

logout();

} else {

alert("Logout canceled.");

}



**🛠️ When to Use:**

* Confirmation before:
  + Deleting something
  + Logging out
  + Navigating away with unsaved changes

**3️] Prompt Box**

**📌 Purpose:**

To **ask the user for input** (like name, number, etc.).

**🧪 Syntax:**

let name = prompt("What is your name?");

**📋 Rules:**

* Returns the input **as a string** (or null if Cancel is clicked).
* Input can be validated with conditionals.

**🎯 Real-World Example:**

let age = prompt("Enter your age:");

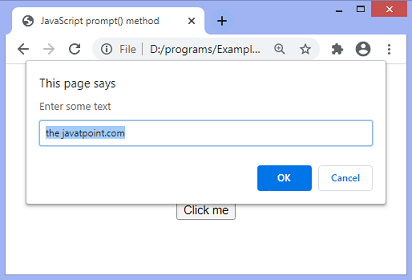
if (age >= 18) {

alert("You are eligible!");

} else {

alert("You must be at least 18.");

}



**🛠️ When to Use:**

* Quick inputs for:
  + Survey questions
  + Verifying email/phone
  + Dynamic greetings

**💡 Why Did These Come into JavaScript?**

* These were introduced early in **browser JavaScript (mid-90s)** for **quick user interaction** before HTML forms or frameworks became popular.
* Still useful for **simple UI tasks** or **demonstrations**, though modern apps now use custom modals.

**🧱 Part 2: JavaScript Methods**

**📌 What is a Method?**

A **method** is a **function associated with an object**.

🔹 In JavaScript, **everything is an object**, so methods can exist on strings, arrays, numbers, etc.

**🔑 Syntax:**

objectName.methodName(arguments)

**🧠 Common Built-in Methods (Grouped by Type)**

**🟠 String Methods**

| **Method** | **Example** | **Output** |
| --- | --- | --- |
| length | "hello".length | 5 |
| toUpperCase() | "hi".toUpperCase() | "HI" |
| includes() | "test".includes("e") | true |
| replace() | "abc".replace("a", "z") | "zbc" |

**🔵 Array Methods**

| **Method** | **Example** | **Output** |
| --- | --- | --- |
| push() | [1, 2].push(3) | Adds 3 |
| pop() | [1, 2].pop() | Removes 2 |
| map() | [1, 2, 3].map(x => x\*2) | [2, 4, 6] |
| filter() | [1,2,3].filter(x => x>1) | [2, 3] |

**🟢 Number Methods**

| **Method** | **Example** | **Output** |
| --- | --- | --- |
| toFixed() | (3.14159).toFixed(2) | "3.14" |
| toString() | (42).toString() | "42" |

**🔴 Math Methods**

| **Method** | **Example** |  |
| --- | --- | --- |
| Math.round(4.7) | → 5 |  |
| Math.random() | → Random value |  |
| Math.max(2, 5, 10) | → 10 |  |

**🎯 Real-World Example: Validate Name Length**

let name = prompt("Enter your name:");

if (name && name.trim().length > 2) {

alert("Welcome " + name.toUpperCase());

} else {

alert("Invalid name!");

}

**🧠 When to Use Methods:**

* Any time you are **working with data types** (Strings, Arrays, etc.)
* Helps make code **clean, readable, and efficient**

**🆚 JavaScript Methods vs Java Methods**

| **Feature** | **JavaScript** | **Java** |
| --- | --- | --- |
| Type | Dynamic | Static (with type declarations) |
| Class Needed? | Not required for built-in types | Must be in a class |
| Flexibility | High | More structured |
| Inheritance | Prototypes | Class-based |

**✅ Summary**

| **Concept** | **Purpose** | **Key Function** |
| --- | --- | --- |
| alert() | Show info | No input, only OK |
| confirm() | Ask user for confirmation | OK / Cancel |
| prompt() | Ask for input | Returns string |
| JS Methods | Work on data types like string, array | map, push, toUpperCase, etc. |

**🌐 What is DOM?**

The **DOM (Document Object Model)** is a **tree-like structure** that represents the HTML elements of a web page. JavaScript allows us to interact with this tree — select elements, change content, add/remove elements, etc.

**📌 Why Do We Need DOM Manipulation?**

DOM manipulation came with early JavaScript to:

* Dynamically **update content** without reloading the page.
* Enable **interactive behavior** (toggle menus, update styles, etc.).
* **React to user actions** like clicks, typing, mouse movement.

**📚 Concept 1: Selecting Elements**

**1️] getElementById()**

**📌 Purpose:**

Select a single element using its **unique ID**.

**🧪 Syntax:**

document.getElementById("myId")

**🎯 Example:**

<p id="greet">Hello</p>

<script>

let element = document.getElementById("greet");

element.style.color = "blue";

</script>

**✅ Use When:**

You know the element has a **unique ID**.

**2️] getElementsByClassName()**

**📌 Purpose:**

Selects **multiple elements** that share the same class.

**🧪 Syntax:**

document.getElementsByClassName("myClass")

**🎯 Example:**

<div class="item">A</div>

<div class="item">B</div>

<script>

let items = document.getElementsByClassName("item");

for (let item of items) {

item.style.fontWeight = "bold";

}

</script>

**✅ Use When:**

You want to style or work with **multiple similar elements**.

**3️] getElementsByTagName()**

**📌 Purpose:**

Selects **all elements** of a specific HTML tag.

**🧪 Syntax:**

document.getElementsByTagName("div")

**🎯 Example:**

<p>Hello</p>

<p>World</p>

<script>

let paragraphs = document.getElementsByTagName("p");

paragraphs[0].style.fontSize = "20px";

</script>

**📚 Concept 2: querySelector() & querySelectorAll()**

**4️] document.querySelector()**

**📌 Purpose:**

Selects the **first element** that matches a **CSS selector**.

**🧪 Syntax:**

document.querySelector("selector")

**🎯 Example:**

<div class="note">Note 1</div>

<div class="note">Note 2</div>

<script>

let note = document.querySelector(".note");

note.style.color = "green"; // Only Note 1

</script>

**✅ Use When:**

You want to apply changes to the **first match** (ID, class, or element).

**5️] document.querySelectorAll()**

**📌 Purpose:**

Selects **all matching elements** using a CSS selector and returns a **NodeList**.

**🧪 Syntax:**

document.querySelectorAll("selector")

**🎯 Example:**

<ul>

<li>Apple</li>

<li>Banana</li>

</ul>

<script>

let items = document.querySelectorAll("li");

items.forEach(item => {

item.style.textTransform = "uppercase";

});

</script>

**✅ Use When:**

You need to **loop through multiple elements** matching a selector.

**📚 Concept 3: Adding Content – append() vs appendChild()**

**6️] appendChild()**

**📌 Purpose:**

Appends a **single node element** as the last child of a parent node.

**🧪 Syntax:**

parent.appendChild(newChildNode)

**🎯 Example:**

<ul id="fruits"></ul>

<script>

let li = document.createElement("li");

li.textContent = "Mango";

document.getElementById("fruits").appendChild(li);

</script>

**✅ Use When:**

You’re adding a **single DOM node** (not text or multiple elements).

**7️] append()**

**📌 Purpose:**

Appends **nodes or strings** (even multiple items).

**🧪 Syntax:**

parent.append("text", childNode)

**🎯 Example:**

let div = document.createElement("div");

div.append("Hello ", document.createElement("span"));

**✅ Use When:**

You want to **append multiple items** or combine **text + elements**.

**🆚 append() vs appendChild()**

| **Feature** | **append()** | **appendChild()** |
| --- | --- | --- |
| Accepts strings | ✅ Yes | ❌ No |
| Accepts multiple args | ✅ Yes | ❌ No (only one node) |
| Returns | undefined | Returns appended node |

**🎯 Real-World Example: Add Item to List on Button Click**

<input type="text" id="taskInput">

<button onclick="addTask()">Add Task</button>

<ul id="taskList"></ul>

<script>

function addTask() {

let task = document.getElementById("taskInput").value;

let li = document.createElement("li");

li.textContent = task;

document.getElementById("taskList").appendChild(li);

}

</script>

**✅ Summary**

| **Concept** | **Purpose** |
| --- | --- |
| getElementById() | Get a single element by ID |
| getElementsByClassName() | Get elements with same class |
| getElementsByTagName() | Get all elements of a tag |
| querySelector() | First match via CSS selector |
| querySelectorAll() | All matches via CSS selector |
| appendChild() | Append one element node |
| append() | Append multiple nodes or text |

**🧠 JavaScript DOM Manipulation Quiz (with Answers)**

**1. What does DOM stand for?**

A. Data Object Model  
B. Document Object Model   
C. Data Oriented Method  
D. Dynamic Output Method

**✔️ Explanation:** DOM stands for *Document Object Model*, which represents the structure of a web page in a way that JavaScript can interact with.

**2. Which of the following is used to select an element by its ID?**

A. document.getElementByClassName()  
B. document.getElementById()   
C. document.querySelectorAll()  
D. document.getElementsByTagName()

**✔️ Explanation:** getElementById() is the correct method for selecting an element using its id attribute.

**3. What does document.querySelectorAll('.item') return?**

A. The first element with class "item"  
B. A NodeList of all elements with class "item"   
C. An array of elements  
D. The last element with class "item"

**✔️ Explanation:** querySelectorAll() returns a **static NodeList** of all matching elements.

**4. True or False:**

append() and appendChild() can be used interchangeably in all cases.

❌ **Answer: False**  
**✔️ Explanation:** appendChild() only accepts nodes, while append() can accept both strings and nodes, making them **not fully interchangeable**.

**5. Which method can select the first <div> on the page?**

A. document.getElementByClassName("div")  
B. document.querySelector("div")   
C. document.querySelectorAll("div")[0]   
D. Both B and C

**✔️ Explanation:** Both querySelector("div") and querySelectorAll("div")[0] return the **first <div>**, but the former is more direct.

**6. What is the difference between append() and appendChild()?**

A. appendChild() can add multiple nodes; append() adds only one  
B. append() can add strings or nodes; appendChild() only nodes   
C. There is no difference  
D. appendChild() is newer than append()

**✔️ Explanation:** append() is more flexible—it can append strings or elements. appendChild() works only with Node objects.

**7. What will this code output?**

const element = document.getElementById("title");

console.log(element.textContent);

A. Prints the ID of the element  
B. Prints the inner HTML  
C. Prints the text content inside the element   
D. Error in console

**✔️ Explanation:** textContent extracts only the text inside the element, without any tags.

**8. Fill in the blank:**

To select all <li> elements with class "menu-item":

const items = document.\_\_\_\_\_\_\_\_\_\_('li.menu-item');

✅ **Answer:** querySelectorAll  
**✔️ Explanation:** The selector 'li.menu-item' uses standard CSS syntax, and querySelectorAll retrieves all matching elements.

**9. What does this code do?**

const p = document.createElement("p");

p.textContent = "Hello World!";

document.body.appendChild(p);

A. Adds "Hello World!" to all <p> elements  
B. Replaces the body content  
C. Creates a new paragraph and adds it to the body   
D. Logs "Hello World!" to console

**✔️ Explanation:** This dynamically creates a new <p> element with content and appends it to the end of the <body>.

**10. True or False:**

document.getElementsByClassName("box") returns an array.

❌ **Answer: False**  
**✔️ Explanation:** It returns an **HTMLCollection**, which is *array-like* but not a real array.

**💡 Project: Live Weather Dashboard**

**✅ Features:**

* City-based weather simulation
* Updates temperature, condition, icon
* Dynamically changes background color based on temperature
* Allows user to change city
* Handles invalid input
* Demonstrates DOM style, content, attribute manipulation

**✅ 1. HTML (weather.html)**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Live Weather Dashboard</title>

<link rel="stylesheet" href="weather.css">

</head>

<body>

<div class="weather-container">

<h1>🌤️ Weather Dashboard</h1>

<input type="text" id="cityInput" placeholder="Enter city name">

<button id="searchBtn">Search</button>

<div id="weatherCard" class="weather-card hidden">

<h2 id="cityName">City Name</h2>

<p id="temperature">Temp: 0°C</p>

<p id="condition">Condition: Clear</p>

<img id="icon" src="" alt="weather icon" />

</div>

<p id="errorMsg" class="error-msg hidden">City not found. Try again!</p>

</div>

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

</body>

</html>

**✅ 2. CSS (weather.css)**

body {

font-family: 'Segoe UI', sans-serif;

background: #dfefff;

margin: 0;

padding: 0;

text-align: center;

}

.weather-container {

margin-top: 50px;

}

input, button {

padding: 10px;

margin: 5px;

font-size: 16px;

}

.weather-card {

border-radius: 8px;

padding: 20px;

display: inline-block;

margin-top: 20px;

background-color: #ffffff;

box-shadow: 0 4px 10px rgba(0,0,0,0.1);

transition: background-color 0.5s ease;

}

img {

width: 80px;

height: 80px;

}

.hidden {

display: none;

}

.error-msg {

color: red;

font-weight: bold;

margin-top: 10px;

}

**✅ 3. JavaScript (weather.js)**

const searchBtn = document.getElementById("searchBtn");

const cityInput = document.getElementById("cityInput");

const weatherCard = document.getElementById("weatherCard");

const cityName = document.getElementById("cityName");

const temperature = document.getElementById("temperature");

const condition = document.getElementById("condition");

const icon = document.getElementById("icon");

const errorMsg = document.getElementById("errorMsg");

// Fake weather data (simulate API)

const weatherData = {

mumbai: { temp: 32, condition: "Sunny", icon: "https://cdn-icons-png.flaticon.com/512/869/869869.png" },

delhi: { temp: 28, condition: "Cloudy", icon: "https://cdn-icons-png.flaticon.com/512/414/414825.png" },

london: { temp: 14, condition: "Rainy", icon: "https://cdn-icons-png.flaticon.com/512/1163/1163624.png" },

newyork: { temp: 9, condition: "Snowy", icon: "https://cdn-icons-png.flaticon.com/512/642/642102.png" }

};

function updateBackground(temp) {

if (temp > 30) {

weatherCard.style.backgroundColor = "#ffe7cc"; // Hot

} else if (temp > 20) {

weatherCard.style.backgroundColor = "#d6f5d6"; // Warm

} else if (temp > 10) {

weatherCard.style.backgroundColor = "#cceeff"; // Cool

} else {

weatherCard.style.backgroundColor = "#e0e0ff"; // Cold

}

}

searchBtn.addEventListener("click", () => {

const city = cityInput.value.toLowerCase().trim();

if (weatherData[city]) {

const data = weatherData[city];

cityName.textContent = city.charAt(0).toUpperCase() + city.slice(1);

temperature.textContent = `Temp: ${data.temp}°C`;

condition.textContent = `Condition: ${data.condition}`;

icon.src = data.icon;

icon.alt = data.condition;

updateBackground(data.temp);

weatherCard.classList.remove("hidden");

errorMsg.classList.add("hidden");

} else {

weatherCard.classList.add("hidden");

errorMsg.classList.remove("hidden");

}

cityInput.value = "";

});

**🔍 JS DOM Manipulation Demonstrated**

| **Feature** | **DOM Methods Used** |
| --- | --- |
| Read user input | input.value |
| Change text dynamically | textContent |
| Show/hide components | classList.add/remove/toggle("hidden") |
| Change image source/alt text | element.src, element.alt |
| Modify styles conditionally | element.style.backgroundColor |
| Handle invalid input | alert or show error with class toggle |
| Dynamically update classes | classList for showing/hiding states |

**🧩 Real-World Assignment: "Dynamic To-Do List Web App"**

**📝 Objective:**

Create a simple web-based **To-Do List** that allows users to:

* Add tasks
* Mark tasks as complete
* Delete tasks
* See total and completed task counts

**✅ Requirements (Mapped to Topics)**

**1. Understanding the DOM & DOM Manipulation**

* Students should explain in comments or a small write-up how the DOM allows them to modify elements in real time.

**2. Selecting and Manipulating Elements**

* Use getElementById, getElementsByClassName, and getElementsByTagName to access:
  + The input box
  + Task list container
  + Buttons

**3. querySelector & querySelectorAll**

* Use querySelector to select a specific button or element with a class.
* Use querySelectorAll to update the task count or apply styles to all completed tasks.

**4. Using append() & appendChild()**

* Dynamically create <li> elements with task details and append them to the <ul> list using appendChild().
* Use append() to optionally add text and buttons together in one step.

**💡 Features to Implement**

* Input field to enter a new task
* “Add Task” button
* Display list of tasks dynamically using JavaScript
* Each task has:
  + Task name
  + “Complete” button to mark the task
  + “Delete” button to remove the task
* Display total number of tasks and how many are completed

**📦 Starter HTML Template (Optional)**

<!DOCTYPE html>

<html>

<head>

<title>To-Do List</title>

</head>

<body>

<h1>My To-Do List</h1>

<input id="taskInput" type="text" placeholder="Enter a new task" />

<button id="addBtn">Add Task</button>

<ul id="taskList"></ul>

<p>Total Tasks: <span id="totalCount">0</span> | Completed: <span id="completedCount">0</span></p>

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

</body>

</html>

**📚 Expected Skills Practiced**

* DOM selection and updates
* Adding/removing nodes dynamically
* Event handling
* Understanding of HTML structure through JavaScript

📂 **File: todo.js**

// Selecting elements using different methods

const taskInput = document.getElementById('taskInput');

const addBtn = document.getElementById('addBtn');

const taskList = document.getElementById('taskList');

const totalCountSpan = document.querySelector('#totalCount');

const completedCountSpan = document.querySelector('#completedCount');

let totalCount = 0;

let completedCount = 0;

// Function to update counters

function updateCounts() {

totalCountSpan.textContent = totalCount;

completedCountSpan.textContent = completedCount;

}

// Add task when button is clicked

addBtn.addEventListener('click', () => {

const taskText = taskInput.value.trim();

if (taskText === '') {

alert('Please enter a task!');

return;

}

// Create new list item

const li = document.createElement('li');

// Create task text node

const taskNode = document.createElement('span');

taskNode.textContent = taskText;

taskNode.className = 'task-text';

// Complete button

const completeBtn = document.createElement('button');

completeBtn.textContent = 'Complete';

completeBtn.className = 'complete-btn';

// Delete button

const deleteBtn = document.createElement('button');

deleteBtn.textContent = 'Delete';

deleteBtn.className = 'delete-btn';

// Append buttons and task to list item

li.append(taskNode, completeBtn, deleteBtn);

// Append list item to task list

taskList.appendChild(li);

// Clear input field

taskInput.value = '';

// Update count

totalCount++;

updateCounts();

// Event: Mark task as complete

completeBtn.addEventListener('click', () => {

if (!taskNode.classList.contains('completed')) {

taskNode.style.textDecoration = 'line-through';

taskNode.classList.add('completed');

completedCount++;

} else {

taskNode.style.textDecoration = 'none';

taskNode.classList.remove('completed');

completedCount--;

}

updateCounts();

});

// Event: Delete task

deleteBtn.addEventListener('click', () => {

if (taskNode.classList.contains('completed')) {

completedCount--;

}

taskList.removeChild(li);

totalCount--;

updateCounts();

});

});