# **Basics of JavaScript**

## Method to include JavaScript

You can include JavaScript in your HTML document in the following ways:

### 1. Inline JavaScript:

### 2. External JavaScript:

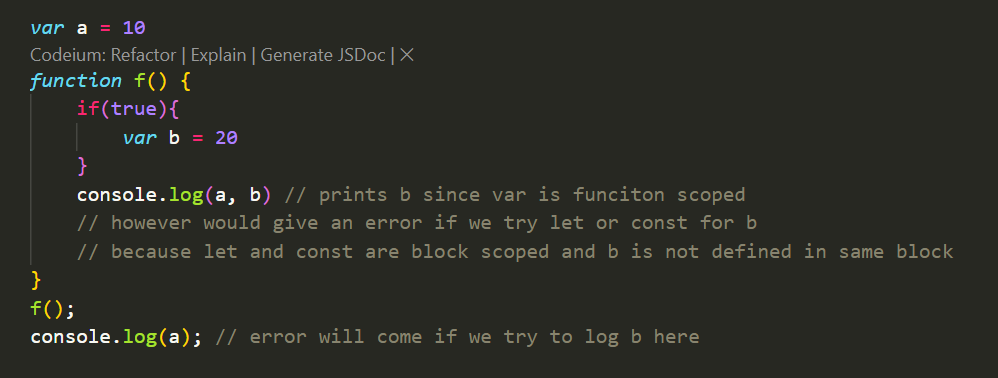
### 3. JavaScript in the <head> or <body>

**<head>:** Generally preferred for scripts that need to run before the page content is loaded (e.g., libraries, setup code).

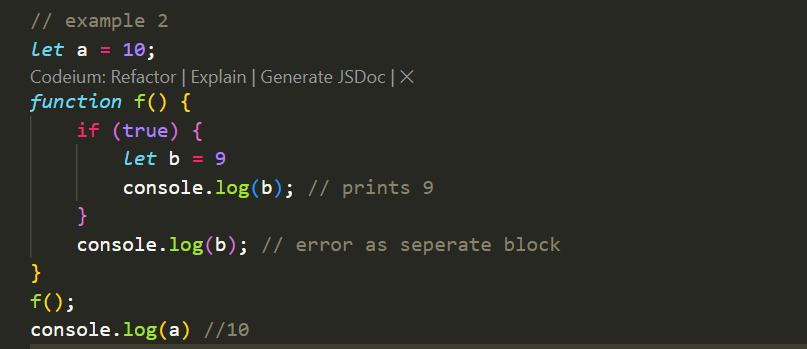
**<body>:** Used for scripts that interact with page elements or need to execute after the page content has loaded.

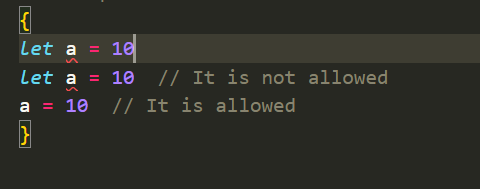
## Var let and const:

### Var:

* It is either function scoped or it is block scoped

### Let:

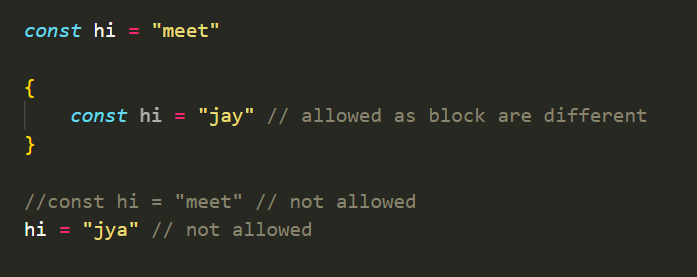
* The let keyword is improved version of var introduced in ES6.
* ****Block scoped
* Cannot be redeclared in same block but can be reassigned in it



* Can be over written in nested blokc when re declared with same name

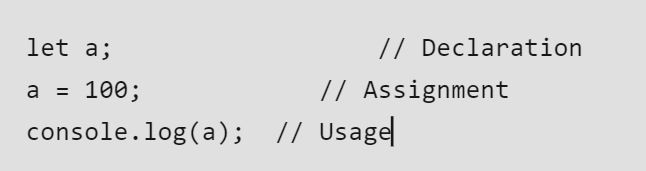
### Const:

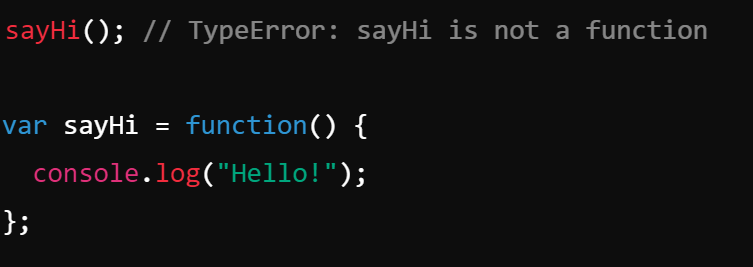
* Very similar to Let
* Only difference is that it cannot be reassigned as well in the same block which we do in case of let. However can be overwritten in nested block



## Hoisting in Javascript

* **JavaScript Hoisting** is the behavior where the interpreter moves function and variable declarations to the top of their respective scope before executing the code.
* However It is very important to note ***JavaScript only hoists declarations, not initializations.***

Variable lifecycle:

* Funcitons are fully hosited with definition and declaration. However not work for arrow functions
* For var, declaration is hoisted with default undefined
* While let and const declarations are hoisted, you cannot access them before their declaration in the code. This period between the start of the scope and the actual declaration is called the **Temporal Dead Zone**.
* **Function expressions** (assigned to variables) are not hoisted.
* 

**let and const**: Hoisted but not initialized. They remain in the **Temporal Dead Zone (TDZ)** until the actual declaration is encountered in the code, and accessing them before initialization causes a **ReferenceError**.

## Basic of Action and Event

**1. Event:**

* An **event** is something that happens in the browser or on the webpage. It could be triggered by user interaction (like clicking, typing, or scrolling) or by the browser itself (such as when a page loads).
* Examples of events:
  + A button is clicked (click event).
  + A key is pressed (keydown or keyup event).
  + The page finishes loading (load event).
* Events are "fired" by the browser, and JavaScript can listen for these events using **event listeners**.

**2. Action:**

* An **action** is what happens **in response** to an event. It’s the code or behavior that gets executed when the event occurs.
* Actions are what you define to handle an event, like running a function to display a message when a button is clicked.
* For example, when the click event occurs, the action could be to submit a form, change the background color, or show an alert message.

**Key Difference:**

* **Event**: The trigger (e.g., a user clicks a button).
* **Action**: The response or behavior (e.g., submit a form when the button is clicked).

## Basic events of javascript

When you store an HTML element in a variable in JavaScript (using methods like getElementById, querySelector, etc.), you can access several important properties that are frequently used in event operations. Here's a list of some of the most useful properties:

**1. innerHTML / innerText**

* **innerHTML**: Gets or sets the HTML content inside an element (including any HTML tags).
* **innerText**: Gets or sets the text content inside an element (ignores any HTML tags).

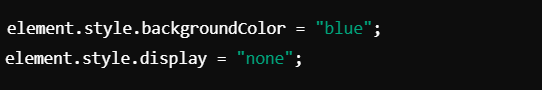
**2. value**

* **value**: Used for form elements (like input, textarea) to get or set the current value of the input field.

**3. classList**

* **classList**: Provides methods to manipulate the CSS classes of an element (e.g., add, remove, toggle).

**4. style**

* **style**: Used to directly modify the inline CSS styles of an element.

**5. id**

* **id**: Returns or sets the id attribute of an element.

**6. attributes**

* **attributes**: Returns a collection of all attributes of an element, allowing you to get or manipulate any attribute.

**7. disabled**

* **disabled**: For input elements, this property is used to disable or enable form controls

**8. checked**

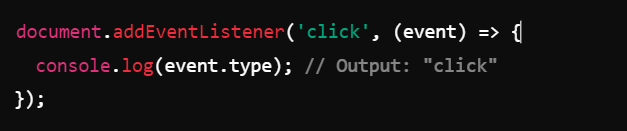
* **checked**: Used for checkbox or radio button elements to determine if they are checked



## Event properties that is passed to callback

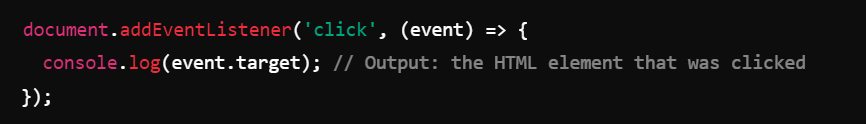
* When an event occurs in JavaScript (e.g., clicking a button, pressing a key), an **event object** is automatically passed to the callback function associated with that event. This event object contains useful information and properties about the event that just happened. Here are some of the most important properties of the event object:

**1. type**

* **Description**: Returns the type of the event (e.g., 'click', 'keyup', 'submit').

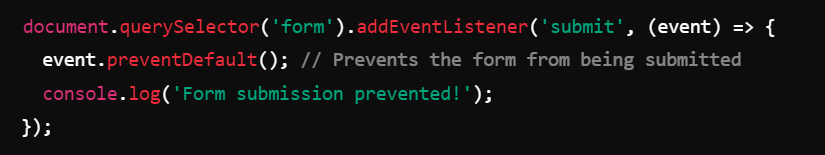
**2. target**

* **Description**: Returns the element that triggered the event (i.e., the event's origin).



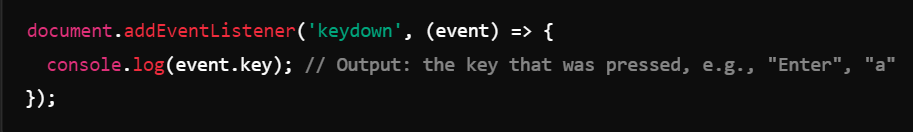
**3. preventDefault()**

* **Description**: Prevents the default action associated with the event (e.g., prevents form submission or following a link).



**4. key (For keyboard events)**

* **Description**: Returns the value of the key that was pressed.



## Local Storage and Session Storage

* Web stosrate API are used by develops to data in the browser and this data is nothing but some key value pairs only.
* Storing data on browser make optimization of web apps
* Now there are to ways to store the data on the web browes:  
  1st : Session Storage
  + Has some advantages over the cookies like
    - Not send to server by request
    - Cookie can store 4000B of data while atleast 5MB can be stored here which is may wore
  + Has some disadvantage also like
    - Are created for a particular session time only
    - Lost when the user leaves the tab

2nd : Local Storage

* + Has some advantages like
    - Remains on the browser even after when the user left the brower
    - Used to store user specific data for request which is more widly used and reduce server burden
    - More storage capacity
    - However the capacity actually depends on the device being used
* **ALL these local storage and session storage follow the same origin policy meaning we can access them on same protocol + host + port**

## Code examples

**What are Local Storage and Session Storage?**

They are both **web storage APIs** that allow you to store data in a user's browser. Unlike **cookies**, they don’t get sent to the server with every request, making them more efficient for storing temporary data.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Local Storage** | **Session Storage** |
| **Data Persistence** | Stored permanently (unless manually cleared) | Removed when the browser tab is closed |
| **Size Limit** | 5-10 MB per domain | 5-10 MB per domain |
| **Accessibility** | Available across all tabs/windows of the same origin | Only available in the tab that created it |
| **Storage Type** | Stores data as key-value pairs in string format | Stores data as key-value pairs in string format |

**1. Local Storage**

* Used for storing **persistent** data in the browser.
* Data **remains even after the user closes the browser** or restarts their device.

**Basic Operations**

**Store Data**

localStorage.setItem("username", "JohnDoe");

**Retrieve Data**

let user = localStorage.getItem("username");

console.log(user); // Output: JohnDoe

**Remove a Specific Item**

localStorage.removeItem("username");

**Clear All Stored Data**

localStorage.clear();

**2. Session Storage**

* Used for storing **temporary** data.
* Data is **cleared when the tab is closed**.

**Basic Operations**

**Store Data**

sessionStorage.setItem("sessionID", "ABC123");

**Retrieve Data**

let session = sessionStorage.getItem("sessionID");

console.log(session); // Output: ABC123

**Remove a Specific Item**

sessionStorage.removeItem("sessionID");

**Clear All Session Data**

sessionStorage.clear();

**3. How to Use Local Storage & Session Storage in jQuery**

Since you're using jQuery, let's integrate it:

**Set and Get Local Storage**

$(document).ready(function() {

// Store data

localStorage.setItem("userEmail", "user@example.com");

// Retrieve data

let email = localStorage.getItem("userEmail");

console.log(email);

});

**Set and Get Session Storage**

$(document).ready(function() {

// Store data

sessionStorage.setItem("cartItems", "5");

// Retrieve data

let cart = sessionStorage.getItem("cartItems");

console.log(cart);

});

**4. Common Use Cases**

|  |  |  |
| --- | --- | --- |
| Scenario | Local Storage | Session Storage |
| **User Authentication (JWT Token, Session ID, User Preferences)** | ✅ Yes | ❌ No (not persistent) |
| **Shopping Cart (for guest users)** | ✅ Yes | ✅ Yes (if you want cart to reset on tab close) |
| **Theme Preferences (Dark/Light Mode)** | ✅ Yes | ❌ No |
| **Temporary Form Data** | ❌ No | ✅ Yes |

**5. Limitations of Web Storage**

* **Cannot store complex data types** (only **strings**).
* **No expiration date** for Local Storage (must be handled manually).
* **Can’t store large data** (max **5-10MB** per domain).
* **Not secure for sensitive data** (e.g., passwords, tokens) since it can be accessed via JavaScript.

**⚠️ Security Tip:** Never store **sensitive user data** (passwords, authentication tokens) in Local Storage because JavaScript can be exploited (XSS attacks).

**6. Alternative Storage Options**

If local/session storage isn't enough, consider:

* **IndexedDB** → For larger, structured data storage.
* **Cookies** → If you need data sent to the server with requests.
* **Server-side Storage (Database, Redis, etc.)** → For secure, persistent data storage.

**🔥 Quick Recap**

* **Local Storage**: Stores data **permanently** (until manually deleted).
* **Session Storage**: Stores data **temporarily** (clears on tab close).
* **Use Cases**: Local Storage for preferences, Session Storage for temporary data.
* **Security Risks**: Don't store sensitive info in either.

## 8.Basics of cookies

* Cookie is **basically set by a web browser using** set cookie HTTP header. So next time when the browser sends the request to the same domain, the cookie is send using the cookie HTTP header so the server knows who send the request !
* We can access the cookie using document.cookie
* Cookies are delimeted by semicolon
* Not : document.cookie = “meet=joshi” does not overwrite all cookie but instead add a new cookie to it.
* Limit of cookie size is 4KB
* Maximum cookie is browswer specific and is around 20
  + Propeties of cookie
    - To delete set expiry to older date
    - Use path to have limited acces routes cookie visible
    - Use uriEncodecompenent to store key and value of any name and use URIdecodecompoent to read back

## OOJS:

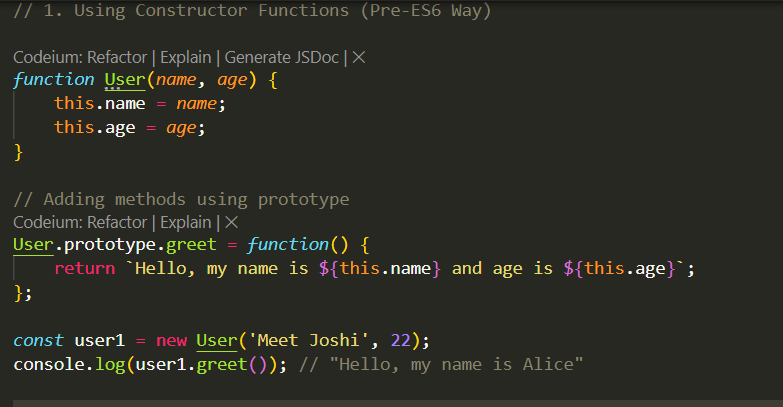
**Object-Oriented JavaScript (OOJS)** refers to the approach of using **Object-Oriented Programming (OOP)** principles in JavaScript. Although JavaScript is primarily a prototype-based language, it supports OOP concepts such as classes, objects, inheritance, encapsulation, and polymorphism.

**Key Concepts of OOJS**

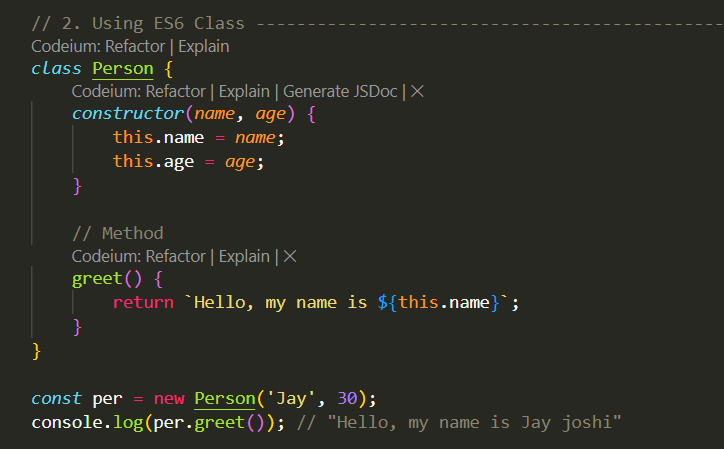
1. **Objects**: In OOJS, objects are instances of classes that contain both data (properties) and behavior (methods). JavaScript uses objects extensively, and nearly everything in JavaScript is an object.
2. **Classes**: Introduced in ES6, classes are a way to define objects and their blueprints.
3. **Encapsulation**: The bundling of data (properties) and methods that operate on the data into a single unit (object or class).
4. **Inheritance**: Allows one class to inherit properties and methods from another class.
5. **Polymorphism**: Allows objects to be treated as instances of their parent class.

Ways to Implement Classes in JavaScript

1. **Using Constructor Functions (Pre-ES6 Way)**



1. **Using ES6 class Syntax**



## Events of JS

**Event Handler Priorities and Execution Phases in JavaScript**

**Priorities of Event Handlers**

When an event triggers in JavaScript, different ways of attaching handlers determine which one executes first. Here’s the hierarchy in order of priority:

1. **addEventListener Method**: Event listeners added with addEventListener take top priority and are processed first.
2. **on Properties**: Handlers assigned to the on properties (like onclick) of elements come next in priority.
3. **HTML Attribute**: Inline attributes within the HTML (like onclick directly in tags) are handled last.

This priority sequence is especially useful when you’re debugging or controlling multiple ways of handling the same event.

**Phases of Event Execution**

Event execution isn’t just a single action; it actually flows through three distinct phases: *Capturing*, *Targeting*, and *Bubbling*.

1. **Capture Phase**:
   * The event starts from the root of the DOM (like html) and travels downward through each nested element until it reaches the one you actually interacted with, the target element.
   * Though it’s often less visible, the capturing phase enables handling events at higher levels before they reach the target.
   * For example, if you clicked on a button inside a div within the body tag, the capture path would be: html -> body -> div -> button
2. **Target Phase**:
   * Once the event reaches the intended target, the target phase begins.
   * This is where handlers tied directly to the target element are executed.
3. **Bubbling Phase**:
   * After reaching the target, the event travels back up the DOM tree through each parent element until it reaches the root.
   * This means that handlers attached to parent elements (like the containing div or body for a button click) will be executed in order on the way back up.
   * In the same example as above, the bubbling path would be: button -> div -> body -> html

**Controlling Event Propagation**

To manage how events propagate, JavaScript provides methods that can halt or customize the natural event flow:

* **Stop Bubbling to Parents**: If you don’t want the event to trigger parent-level handlers during the bubbling phase, you can use:

event.stopPropagation();

* **Prevent Other Handlers from Firing**: When you have multiple handlers attached to the same element, and you want to stop the execution of other handlers after a specific one is triggered, use:

event.stopImmediatePropagation();