# Notes

* **3xx status codes** are specifically used for redirection.
* **307 (Temporary Redirect)** indicates that the request should be repeated with another URI, but future requests should still use the original URI.

Let's briefly look at the others:

* **2xx status codes** indicate success.
  + **202 (Accepted)** means the request has been accepted for processing, but the processing has not been completed.
  + **226 (IM Used)** means the server has fulfilled a GET request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.
* **5xx status codes** indicate server errors.
  + **501 (Not Implemented)** means the server does not support the functionality required to fulfill the request.

2. Which operator is used for strict comparison in JS?

A) =  
B) ==  
C) ===  
D) :=

* A) ❌ Assignment, not comparison
* B) ❌ Weak comparison (does type conversion) Loose equality
* C) ✅ Correct – Checks both value and type
* D) ❌ Not valid in JavaScript

**typeof ("John"+"Doe") // Returns "string"**

**typeof 3.14 // Returns "number"**

**typeof NaN // Returns "number"**

**typeof 1234n // Returns "bigint"**

**typeof true // Returns "boolean"**

**typeof {name:'John'} // Returns "object"**

**typeof [1,2,3,4] // Returns "object"**

**typeof {} // Returns "object"**

**typeof [] // Returns "object"**

**typeof null // Returns "object"**

**typeof new Object() // Returns "object"**

**typeof new Array() // Returns "object"(new se jo bhi hoga vo object ho ga)**

**typeof function () {} // Returns "function"**

**typeof x // Returns "undefined"**

**4. Which of the following is not a primitive data type?**  
A) Number  
B) Object  
C) Boolean  
D) Undefined

* A) ❌ Primitive
* B) ✅ Correct – Object is non-primitive
* C) ❌ Primitive
* D) ❌ Primitive

✅ **Correct Answer: B – Object is not a primitive; it's a reference type**

**5. What will be the result of true + true in JS?**  
A) NaN  
B) false  
C) 2  
D) "truetrue"

* A) ❌ JS converts true to 1
* B) ❌ Not a comparison
* C) ✅ Correct – true + true → 1 + 1 = 2
* D) ❌ JS doesn't auto-convert Booleans to strings here

✅ **Correct Answer: C – JS treats true as 1 when using arithmetic operators**

**🔍 What does "NaN" mean in JavaScript?**

NaN stands for **"Not-a-Number"**.  
Despite its name, **NaN is actually of type "number"**.

**Occur as**  the result of an operation that **doesn’t produce a valid number**

**Recap:**

| **Expression** | **Result** | **Explanation** |
| --- | --- | --- |
| "abc" / 2 | NaN | Invalid math operation |
| typeof NaN | "number" | NaN is still a numeric type |
| isNaN(NaN) | true | NaN is indeed Not-a-Number |
| NaN === NaN | false | NaN is **not equal** to itself |

**What is the result of true + false?**

A) NaN  
B) 1  
C) truefalse  
D) undefined

**Answer: B** – true is 1, false is 0. So 1 + 0 = 1.

**What will typeof undefined return? 🡺undefined is its own type.**

**9. Which one of these is NOT a valid JavaScript data type?**

**A) Symbol  
B)** Float **C) Undefined  
D) Boolean**

**Answer: B – JavaScript uses Number for all numeric types, no Float type.**

**typeof function() {} return?==> function**

**11. JavaScript is:**

**A) Compiled language  
B) Interpreted language  
C) Procedural only  
D) None of the above**

**Answer: B – It is interpreted(code is executed Line by Line) (though modern engines use JIT(Just in time) for performance).**

**const variables cannot be re-declared or reassigned.**

**What does “Falsy” mean in JavaScript?**

**A falsy value is a value that is considered false when evaluated in a Boolean context — such as in an if statement.**

**✅ Falsy values in JavaScript:**

**There are exactly 7 falsy values:**

1. **false**
2. **0**
3. **"" (empty string)**
4. **null**
5. **undefined**
6. **NaN**
7. **document.all (rare, legacy quirk)**

**What is hoisting in JavaScript?**

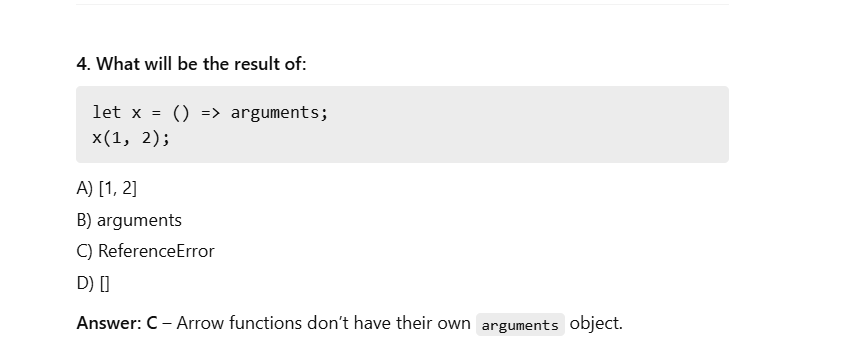
**Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their containing scope (either function or global) during the compile phase before the code is executed.**

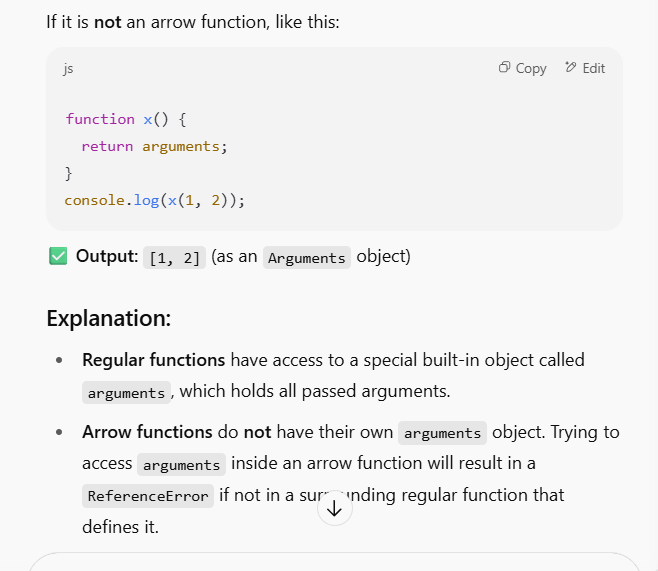
****

**21. What happens if you try to access a let variable before it's declared?**

**A) It returns undefined  
B) It throws a ReferenceError  
C) It returns null  
D) It assigns a default value**

**Answer: B – Accessing a let or const variable before its declaration causes a ReferenceError due to the Temporal Dead Zone. But for Var it will be undefined**

****

****

**18. What is hoisting in JavaScript?**

**A) Variable destruction  
B) Pre-declaration of variables/functions 🡺ans  
C) Looping mechanism  
D) DOM Traversal**

**20. How do you check if a variable is an array?**

**A) typeof var === "array"  
B) var instanceof Array  
C) Array.isArray(var)  
D) Both B and C**

**Answer: D – Both are valid.**

**In the global scope, this refers to the global object**

**console.log(typeof Infinity)🡺Number**

**JSON.parse() converts a JSON string to a JavaScript object.**

****

**31. What does the !! operator do in JavaScript?**

**A) Inverts the value  
B) Converts value to Boolean  
C) Converts Boolean to number  
D) Negates twice and returns original**

**Answer: B – !!value forces any value to a Boolean (true or false).**

**32. Which of the following does NOT create a new scope?**

**A) function() { }  
B) if() { }  
C) for() { }  
D) let block = { }**

**Answer: D – Object literals do not create scope. function, if, for blocks do with let/const.**

**33. What is the result of: [] + {}?**

**A) "[object Object]"  
B) {}  
C) Error  
D) undefined**

**Answer: A – It becomes an empty array + object → coerced to string: \"[object Object]\".**

**34. What is the result of: {} + []?**

**A) 0  
B) "[object Object]"  
C) NaN  
D) ""**

**Answer: A – When {} is interpreted as a block and +[] becomes 0.**

**35. What is the difference between null and undefined?**

**A) undefined is assigned by JavaScript automatically, null is assigned manually  
B) Both are the same  
C) null means not declared  
D) undefined is an object**

**Answer: A – undefined = default uninitialized value, null = intentional empty value.**

# JavaScript Functions, Objects, and OOP Concepts

## Function

function greet() { ... } // Function declaration

const greet = function() { ... } // Function expression

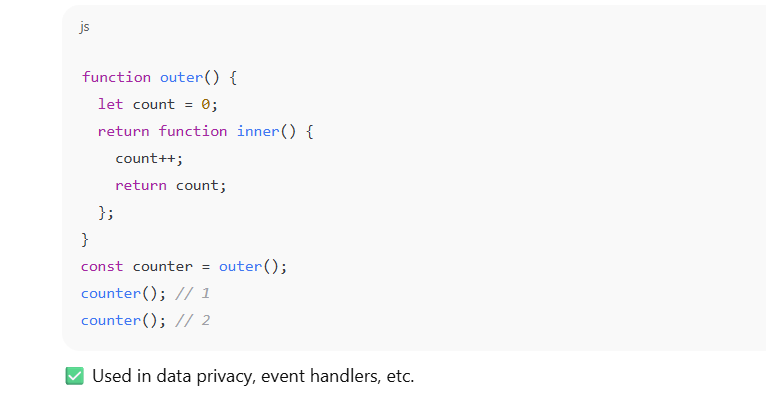
const greet = () => { ... } // Arrow function (ES6)

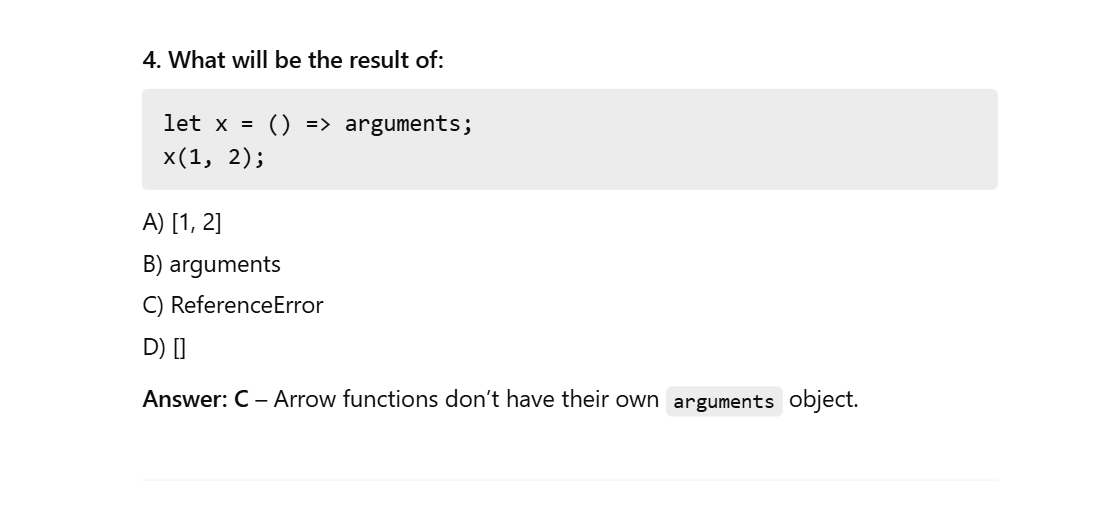
**Function Features:**

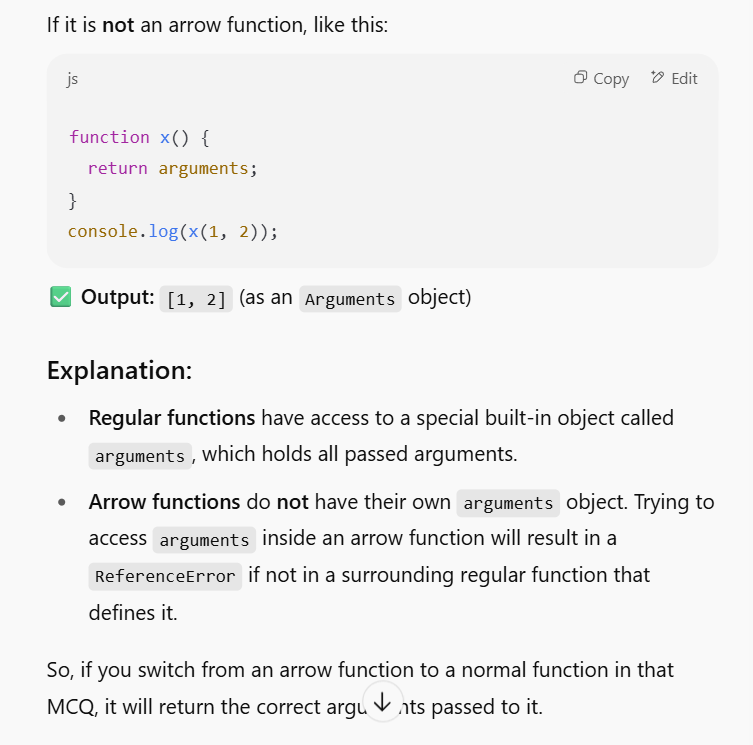
* Can return values using return
* Can be passed as arguments to other functions (First-class citizens)
* Support closures and scoping

### 🧠 Closures (VERY IMPORTANT)

A closure is created when a function "remembers" variables from its lexical scope even when called outside that scope.



****

****

**✅ What is a Closure?**

**A closure is created when a function is defined inside another function and retains access to the outer function’s variables—even after the outer function has finished executing.**

****

**By using closures🡺can you create a private variable in JS🡺Closures allow data to be hidden from outer scope**

🡺Arrow functions do not have their own this🡺They inherit this from parent scope

🡺add a method to an object🡺 object.method = function() {}

🡺inheritance in JavaScript was primarily implemented using constructor functions and their prototype property/(object)

🡺Object.assign()🡺It copies properties shallowly. i,e Shallow copy

**15. What is returned if a constructor function does not explicitly return anything?**

A) undefined  
B) this ==correct  
C) null  
D) window

When a constructor function is invoked with the new keyword, it implicitly returns the newly created instance of the object (which this refers to inside the constructor) if no explicit return statement is present, or if an explicit return statement returns a non-object primitive value (like undefined, null, a number, or a string).

**12. What is prototype chaining?**

**A) A way to override functions  
B) Linking object properties  
C) Inheriting properties through \_\_proto\_\_  
D) None**

**Prototype chaining is a fundamental mechanism in JavaScript for inheritance. When you try to access a property or method on an object, JavaScript first looks for it directly on that object. If it doesn't find it there, it then looks at the object's prototype (which is accessible via the \_\_proto\_\_ internal property or Object.getPrototypeOf()). This process continues up the "chain" of prototypes until the property is found or the end of the chain ( null) is reached.**

**Let's quickly look at why the others aren't the best fit:**

* **A) A way to override functions: While prototype chaining plays a role in how overridden functions are resolved (the one closer in the chain is found first), it's not *itself* the act of overriding. Overriding is when a child object defines a method with the same name as a method in its parent.**
* **B) Linking object properties: While properties are linked, "prototype chaining" specifically describes the *mechanism* of how that linkage enables inheritance and property lookup.**

**✅ JavaScript Summary – Functions, Objects, and OOP Concepts**

**🔹 1. Functions**

* **Declared using the function keyword or as arrow functions (=>).**
* **Can be named, anonymous, or expressions.**
* **Functions are first-class citizens in JavaScript — they can be passed around like variables.**
* **arguments object is available in regular functions (not in arrow functions).**

**🔹 2. Arrow Functions**

* **Introduced in ES6.**
* **Do not have their own this, arguments, or super.**
* **Inherit this from the surrounding lexical context.**
* **Cannot be used as constructors (i.e., with new).**
* **Great for short callbacks, especially where you don’t need this.**

**🔹 3. Closures**

* **A closure is a function that remembers variables from its outer scope even after that outer function has finished executing.**
* **Used to create private variables.**
* **Common in factory functions and callbacks.**

**🔹 4. Objects**

* **Created using object literals ({}), constructors, or Object.create().**
* **Properties can be added dynamically.**
* **Methods are functions assigned as properties of objects.**

**🔹 5. Prototypes and Inheritance**

* **Every JavaScript object has a hidden [[Prototype]] (accessible via \_\_proto\_\_ or Object.getPrototypeOf()).**
* **Prototypes allow inheritance — an object can use methods/properties defined in another object.**
* **Prototype chaining means the lookup for properties/methods goes up the prototype chain until found or null.**

**🔹 6. Classes and ES6 OOP**

* **Introduced with class keyword in ES6.**
* **constructor() defines the initialization logic.**
* **Can have methods, static methods, and use inheritance via extends.**
* **Use super() to call the parent class constructor.**
* **Support core OOP principles: encapsulation, inheritance, polymorphism, and abstraction (conceptually).**

**🔹 7. this Keyword**

* **In regular functions, this refers to the calling object.**
* **In arrow functions, this is lexically bound (based on the surrounding scope).**
* **In global scope (non-strict mode), this refers to the window object.**

**🔹 8. Object Utility Methods**

* **Object.assign() → performs a shallow copy of properties.**
* **Object.keys(), Object.values(), and Object.entries() help iterate over object data.**
* **Object.freeze() makes an object immutable.**

# DOM and Events

🡺querySelector() returns the first match for the selector. i,e **document.querySelector("#demo")**

**Concept Summary: DOM and Events**

**📌 Important Points to Remember**

* The DOM is a structured representation of HTML documents as a tree of objects.
* getElementById() is the most common method to access elements.
* querySelector() is more flexible with CSS-style selectors.
* Use innerHTML to change content, but be cautious (can inject scripts).
* Always attach event listeners using addEventListener().
* Use event.preventDefault() to stop the browser's default action.
* Use event.stopPropagation() to stop bubbling to parent elements.
* DOM elements are JavaScript objects and can be manipulated in real time.
* document.createElement() creates but does not add the element to the page.
* Use appendChild() or insertBefore() to place new elements into the DOM.
* Event propagation flows in two phases: capturing and bubbling.
* Avoid using inline onclick attributes in HTML for better separation of logic.
* Keyboard and mouse events can greatly enhance interactivity.
* The DOM API works with live data, changes reflect immediately on the page.
* Always validate element existence (if (element)) before accessing/modifying it.

**🔍 Concept Summary: DOM and Events**

**🧠 1. What is the DOM?**

* The **Document Object Model (DOM)** is a tree-like structure that represents the contents of an HTML document.
* Every HTML tag is a **node** (element node), and nodes can have attributes (attribute nodes), and content (text nodes).

**🛠️ 2. Accessing Elements**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| getElementById() | Access by ID | document.getElementById("id") |
| getElementsByClassName() | Access by class | document.getElementsByClassName("cls") |
| getElementsByTagName() | Access by tag | document.getElementsByTagName("p") |
| querySelector() | First match using CSS selector | document.querySelector(".my-class") |
| querySelectorAll() | All matches | document.querySelectorAll("div") |

**✏️ 3. Changing Elements**

* element.innerHTML – Gets or sets the HTML content inside an element.
* element.textContent – Gets or sets text only.
* element.setAttribute(name, value) – Sets an attribute value.
* element.style.property = value – Sets a style directly.

**🧱 4. Creating & Inserting Elements**

* document.createElement("tag") – Creates a new element in memory.
* parent.appendChild(child) – Inserts the new element into the DOM.
* parent.insertBefore(newEl, existingEl) – Inserts before a specific child.

**📡 5. Events in JavaScript**

* DOM events are actions like click, hover, keydown, etc.
* You can assign events directly or with addEventListener():

🡺btn.addEventListener("click", () => alert("Clicked!"));

**📦 6. Event Object**

* Passed automatically to the handler function.
* Contains info such as:
  + event.target: the actual element clicked
  + event.type: the event type (click, keyup, etc.)
  + event.clientX, event.clientY: mouse position

**🔄 7. Event Propagation**

* **Bubbling** (default): Event moves up from the target to ancestors.
* **Capturing** (optional): Moves top-down before hitting the target.
* You can control this using a third argument in addEventListener(event, handler, true).

**🚫 8. Event Control**

* event.preventDefault() – Prevents default behavior (e.g., form submission, link click).
* event.stopPropagation() – Prevents bubbling up the DOM.

**📌 Important Points to Remember**

* DOM is a tree-like API used by JavaScript to interact with HTML.
* Always check null before using element references to avoid errors.
* Use querySelector for powerful and flexible element selection (CSS-style).
* Always prefer addEventListener() over inline event attributes like onclick.
* Use innerHTML with caution — it can expose your app to security risks (XSS).
* stopPropagation() and preventDefault() are essential in event control. .
* DOM methods return live or static collections — understand the difference
* Events propagate in two phases — capture (top-down) and bubble (bottom-up).
* Use textContent for text-only updates — safer than innerHTML.

# ✅ Forms and Form Validation in JavaScript

**📌 What Are HTML Forms?**

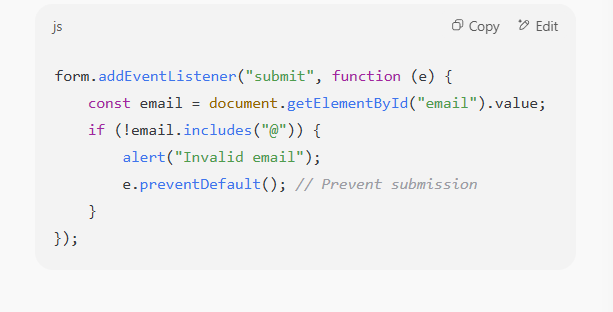
HTML forms are used to collect user input and send it to a server. They use elements like <form>, <input>, <textarea>, and <button>.

**📋 Key HTML Input Attributes:**

* type: Specifies the type of input (text, email, password, etc.)
* required: Ensures the field must be filled before form submission
* pattern: Defines a regex pattern that the input must match
* minlength / maxlength: Restrict the number of characters
* disabled & readonly: Prevent user interaction or editing

**🛠 JavaScript Form Handling:**

* Use document.getElementById("id") to get an input field
* .value retrieves the user-entered data
* Use form.addEventListener("submit", function(e){ ... }) to validate on submission
* e.preventDefault() stops form from submitting

**✅ HTML5 Native Validation:**

* + **Built-in attributes like required, type, pattern automatically validate form inputs**
  + **Adding novalidate to <form> disables this native check**
  + **Native tooltips appear when form input is invalid**

**🧪 JavaScript-Based Validation:**

* + **Offers flexibility beyond HTML5 validation**

**Regular Expressions (RegEx) are used for pattern matching**

* + **Use .setCustomValidity("message") to show custom error messages**
  + **Prevent invalid form submission using event.preventDefault()**

**📌 Best Practices:**

* + **Combine both HTML5 and JavaScript validations for better UX**
  + **Provide real-time feedback to users**
  + **Sanitize and re-validate data on the server to prevent security issues**
  + **Use .reset() to clear form values**
  + **Use .submit() to programmatically trigger form submission**

**🔧 Key HTML Form Elements**

| **Element** | **Purpose** |
| --- | --- |
| <form> | The container that groups input fields |
| <input> | Text fields, checkboxes, radio buttons, buttons |
| <textarea> | Multi-line text |
| <select> & <option> | Dropdown menu |
| <button> | Form submission or custom actions |
| name, id, value, type, placeholder, required | Common attributes |

**✅ JSON & jQuery – Concept Summary**

**🔗 JSON (JavaScript Object Notation)**

* **Lightweight data-interchange format used for transmitting data.**
* **Data is written in key-value pairs.**
* **Supports strings, numbers, arrays, booleans, null, objects.**
* **Syntax resembles JavaScript object literals.**

**Example:**

**{**

**"name": "Shivam",**

**"age": 25,**

**"skills": ["HTML", "CSS", "JS"]**

**}**

**🔍 JSON Key Points Used in MCQs:**

* **Valid keys must be strings wrapped in double quotes.**
* **JSON can be parsed into JavaScript using JSON.parse().**
* **JavaScript objects can be stringified to JSON using JSON.stringify().**
* **Does not support functions or comments.**

****

**💡 jQuery – Concept Summary**

**📌 What is jQuery?**

* **A fast, small, and feature-rich JavaScript library.**
* **Simplifies tasks like DOM manipulation, event handling, animation, and AJAX.**
* **Syntax is concise and cross-browser compatible.**

**🔑 Core Concepts:**

* **Selectors: Begin with $() — selects DOM elements.**

**$("#id"), $(".class"), $("element")**

* **Events: Handle events like click, dblclick, hover, submit.**

**$("button").click(function() { alert("Clicked"); });**

* **Effects & Animations: hide(), show(), toggle(), fadeIn(), slideDown(), etc.**
* **DOM Manipulation:**
  + **html(), text(), val(), attr(), css() to get/set content/attributes/styles**
* **Traversing:**
  + **parent(), children(), siblings(), find(), closest()**
* **AJAX Integration:**
  + **$.get(), $.post(), $.ajax() for asynchronous server communication**
* **Chaining:**
  + **jQuery allows chaining multiple methods:**

$("p").css("color", "red").slideUp().slideDown();

**🔐 Additional Notes:**

* **jQuery is widely supported but often replaced by native JS in modern frameworks (React, Angular).**
* **jQuery plugins extend functionality (e.g., sliders, validation tools).**
* **Lightweight and ideal for simpler projects or when backward compatibility is needed.**

**✅ Key Concepts Reflected in MCQs:**

* **Selectors and their specificity**
* **Event binding and delegation**
* **Common animations and visibility methods**
* **Form manipulation and data access**
* **Traversing methods and chaining behavior**
* **AJAX request handling in jQuery**

**✅ AJAX & Axios – Concept Summary**

**🔹 What is AJAX?**

**AJAX (Asynchronous JavaScript and XML) is a technique that allows web applications to send and receive data from a server asynchronously without reloading the entire page.**

**🔑 Core Concepts:**

* **Uses browser-built-in objects like XMLHttpRequest or fetch().**
* **Communicates via HTTP methods: GET, POST, PUT, DELETE.**
* **Common data formats: JSON (most popular), XML, plain text.**
* **AJAX allows partial updates, leading to faster, more dynamic UIs.**

**🧠 Key Properties of XMLHttpRequest:**

* **readyState: 0 to 4 (4 = done)**
* **status: HTTP status codes (200 = OK, 404 = Not Found, 500 = Server Error)**
* **onreadystatechange: Event triggered during request lifecycle**

**Example:**

**let xhr = new XMLHttpRequest();**

**xhr.open("GET", "data.json", true);**

**xhr.onreadystatechange = function () {**

**if (xhr.readyState == 4 && xhr.status == 200) {**

**console.log(JSON.parse(xhr.responseText));**

**}**

**};**

**xhr.send();**

**🔹 What is fetch()?**

**fetch() is a modern replacement for XMLHttpRequest, returns a Promise.**

fetch("data.json")

.then(response => response.json())

.then(data => console.log(data))

.catch(err => console.error(err));

* **Supports async/await**
* **Requires manual error handling for non-2xx statuses**

**🔹 What is Axios?**

**Axios is a promise-based HTTP client library for the browser and Node.js.**

**✨ Axios Features:**

* **Automatically transforms JSON**
* **Handles request/response interceptors**
* **Supports async/await**
* **Works in both browser & Node.js**
* **Allows setting global headers**

**axios.get('/api/data')**

**.then(res => console.log(res.data))**

**.catch(err => console.error(err));**

**🔧 Axios Key Functions:**

* **axios.get(url)**
* **axios.post(url, data)**
* **axios.put(url, data)**
* **axios.delete(url)**

**🔒 Axios Extras:**

* **Interceptors – Modify request/response globally**
* **CancelToken – Cancel requests**
* **TransformResponse – Customize response processing**

**🔐 HTTP Methods Overview:**

| **Method** | **Purpose** | Idempotent? |
| --- | --- | --- |
| **GET** | **Retrieve data** | **✅ Yes** |
| **POST** | **Create data** | **❌ No** |
| **PUT** | **Replace data** | **✅ Yes** |
| **PATCH** | **Update partial** | **❌ No** |
| **DELETE** | **Delete data** | **✅ Yes** |

**🌐 HTTP Status Codes:**

| **Code** | **Meaning** |
| --- | --- |
| **200** | **OK (success)** |
| **201** | **Created** |
| **204** | **No Content** |
| **301** | **Moved Permanently** |
| **400** | **Bad Request** |
| **401** | **Unauthorized** |
| **403** | **Forbidden** |
| **404** | **Not Found** |
| **500** | **Internal Server Error** |

**🧪 Promises and async/await in AJAX:**

**async function loadData() {**

**try {**

**let response = await axios.get("/api/info");**

**console.log(response.data);**

**} catch (error) {**

**console.error("Failed to fetch:", error);**

**}**

**}**

**✅ Key Concepts Reflected in MCQs:**

* **Parsing JSON with JSON.parse() / JSON.stringify()**
* **Axios request structure and HTTP methods**
* **Status code meanings**
* **Usage of headers like Content-Type, Authorization**
* **Benefits of Promises & async/await**
* **Use of Axios interceptors and cancel tokens**
* Response handling with then(), catch(), finally()

**Node.js – Complete Conceptual Summary**

**📌 What is Node.js?**

* **Node.js is an open-source, cross-platform runtime environment.**
* **It uses Google’s V8 JavaScript engine to run JavaScript code outside the browser.**
* **It is known for its non-blocking, event-driven I/O model which makes it ideal for real-time, data-intensive applications.**

**🧱 Core Architecture and Execution**

* **Single-threaded event loop model.**
* **Built on asynchronous I/O using callbacks, Promises, and async/await.**
* **Excellent for I/O-bound tasks like APIs, database access, and file I/O.**

**📁 Module System (CommonJS)**

* **Node.js uses CommonJS module system.**
* **To import:**

**const fs = require('fs');**

* **To export:**

**module.exports = myFunction;**

* **Modules are local-scoped — variables declared in one module don’t pollute others.**

**🧰 Built-in Core Modules**

* **fs – File System operations**
* **http – Web server creation**
* **url – URL parsing**
* **path – File path utilities**
* **events – Event-driven programming**
* **os – System-level info (CPU, RAM)**
* **stream – For reading/writing continuous data**
* **zlib – Compression (gzip)**
* **crypto – Encryption and hashing**

**📄 Globals in Node.js**

* **\_\_dirname: Directory path of current module**
* **\_\_filename: File path of current module**
* **process: Gives access to env variables, exits, etc.**
* **setTimeout, setInterval, setImmediate, process.nextTick(): All used for scheduling async execution**

**🧾 NPM & Package Management**

* **npm: Node Package Manager**
* **package.json: Metadata and dependencies**
* **package-lock.json: Locks exact versions**
* **Install packages:**

**bash**

**CopyEdit**

**npm install express**

* **Scripts:**

**json**

**CopyEdit**

**"scripts": {**

**"start": "node index.js"**

**}**

**🌐 Creating a Basic Server**

**js**

**CopyEdit**

**const http = require('http');**

**http.createServer((req, res) => {**

**res.end("Hello World");**

**}).listen(3000);**

**⚙️ Event Loop and Asynchronous Code**

* **Node.js is non-blocking and async by nature.**
* **Use of callbacks, Promises, and async/await for I/O handling.**
* **process.nextTick() schedules microtasks.**

**📤 File System (fs) Module**

* **Reading files:**

**js**

**CopyEdit**

**fs.readFile('file.txt', callback);**

* **Writing files:**

**js**

**CopyEdit**

**fs.writeFile('file.txt', 'data', callback);**

* **Streams: fs.createReadStream() and fs.createWriteStream() for large files.**

**📦 Buffer & Stream**

* **Buffer: Temporary storage of binary data.**
* **Stream: Reading/writing files piece by piece.**
  + **Readable, Writable, Duplex, and Transform**

**🔐 Events and EventEmitter**

**js**

**CopyEdit**

**const EventEmitter = require('events');**

**const emitter = new EventEmitter();**

**emitter.on('start', () => console.log("Started"));**

**emitter.emit('start');**

**🌍 HTTP & Web Server**

* **Native http module can serve static/dynamic content.**
* **Frameworks like Express.js simplify routing and middleware logic.**

**🔄 Child Processes**

* **child\_process module lets you run shell commands or other programs.**
* **Methods:**
  + **exec(), spawn(), fork()**

**🧪 JSON Handling**

* **Parse JSON:**

**js**

**CopyEdit**

**JSON.parse(jsonString)**

* **Stringify JS object:**

**js**

**CopyEdit**

**JSON.stringify(obj)**

**📊 Miscellaneous Concepts**

* **typeof null returns "object" (quirk)**
* **typeof NaN is "number"**
* **setTimeout(fn, 0) still waits till current execution completes (Event Loop)**
* **async/await must be inside async function**
* **finally() always executes after .then() or .catch()**
* **axios and fetch are HTTP libraries — axios supports interceptors, fetch is native**

**Streams & Buffers**

* **Stream: Continuous data (like video/files)**
* **Types: Readable, Writable, Duplex, Transform**
* **Buffer: Raw binary data container**

## ✅ Express.js – Concept Summary

**🔹 What is Express.js?**

* Express.js is a **minimal, flexible Node.js web application framework**.
* It provides tools to build **RESTful APIs**, **web servers**, and **middleware-based applications**.
* Built on top of Node’s native http module.

**🔧 Key Features of Express**

| **Feature** | **Description** |
| --- | --- |
| Minimal | Adds just what you need; you build on top of it |
| Middleware | Functions that execute before the final route handler |
| Routing | Define URL-based request handlers (GET, POST, etc.) |
| Templating | Supports template engines like EJS, Pug |
| RESTful API | Easily build CRUD APIs using route handlers |

**📁 Setting Up Express**

npm install express

**Basic Express App**

const express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Hello from Express!');

});

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

**🚦 HTTP Methods**

| **Method** | **Use** |
| --- | --- |
| GET | Fetch data |
| POST | Submit new data |
| PUT | Update existing data |
| DELETE | Remove data |

app.post('/users', (req, res) => { ... });

**🧩 Middleware**

**✅ What is Middleware?**

* **Functions that have access to req, res, and next.**
* **Can modify requests, perform auth, log data, or handle errors.**

(req, res, next)

* req: request object
* res: response object
* next(): pass control to next middleware

**Example:**

app.use((req, res, next) => {

console.log(`${req.method} ${req.url}`);

next();

});

**✅ Built-in Middleware**

* express.json() – Parses JSON body
* express.urlencoded() – Parses URL-encoded data (e.g., form data)
* express.static('folder') – Serves static files like CSS, JS

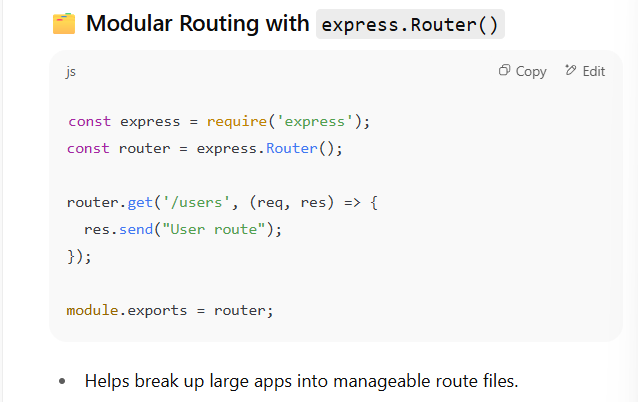


**📥 Handling Form & JSON Data**

app.use(express.urlencoded({ extended: true }));

app.use(express.json());

**🗂 Routing**



**🧩 Template Engines**

* Express supports engines like **Pug**, **EJS**, **Handlebars**

app.set('view engine', 'ejs');

**📦 Serving Static Files**

app.use(express.static('public'));

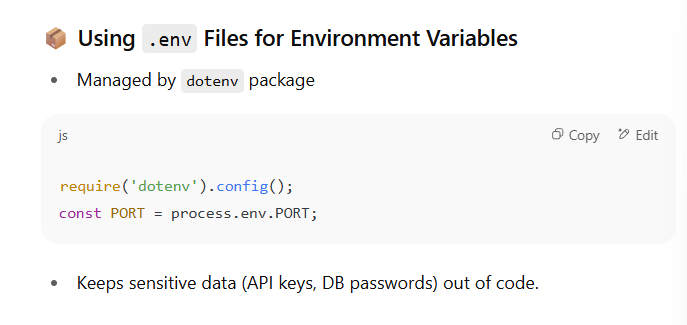
**✋ Error Handling**

app.use((err, req, res, next) => {

console.error(err.stack);

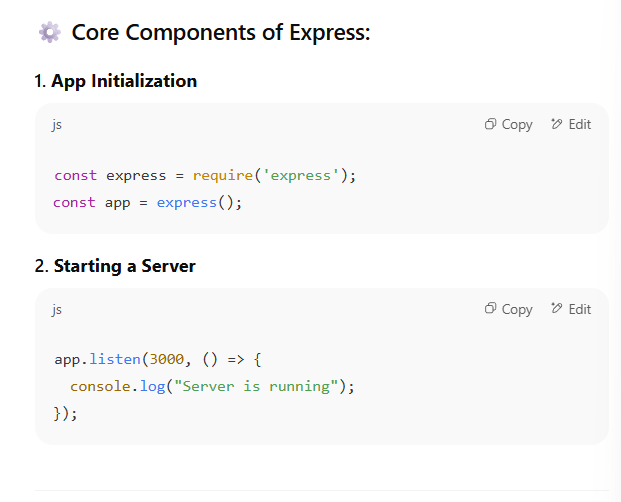
res.status(500).send('Something broke!');

});



**🔐 Summary: Key Concepts**

* Express simplifies server creation and API handling in Node.js.
* Uses routing and middleware system to structure your app.
* Easily handles static files, form data, JSON, and template rendering.



**9. What is the default port often used in local Express servers?  
A) 3000  
B) 8080  
C) 3306  
D) 5000  
✅ Answer: A – Port 3000 is commonly used in development.**

**17. What method is used to define an error-handling middleware?**  
A) app.use(err, req, res)  
B) app.catch(req, res, next)  
C) app.error(req, res, next)  
D) app.use((err, req, res, next) => {})  
✅ **Answer: D** – Error middleware has 4 args: err, req, res, next.

**. What is the purpose of req.body?**  
A) To get headers  
B) To get cookies  
C) To access form/JSON data  
D) To access URL params  
✅ **Answer: C** – req.body contains data sent in the request.

**. Which method renders a view with EJS?  
A) res.view()  
B) res.load()  
C) res.render()  
D) res.ejs()  
✅ Answer: C – Use res.render('viewname') for templat**

**What does req.params access?  
A) Query string parameters  
B) Path parameters in the URL  
C) Form data  
D) Response headers  
✅ Answer: B – req.params captures parameters like /user/:id.**

**What is the use of next() in middleware?  
A) Stops execution  
B) Ends the server  
C) Passes control to next middleware  
D) Restarts the server  
✅ Answer: C – next() passes control to the next middleware or route.**

**What is required to parse form data in Express?  
A) express.static()  
B) express.urlencode()  
C) express.urlencoded()  
D) formParser()  
✅ Answer: C – Use express.urlencoded({ extended: true }).**

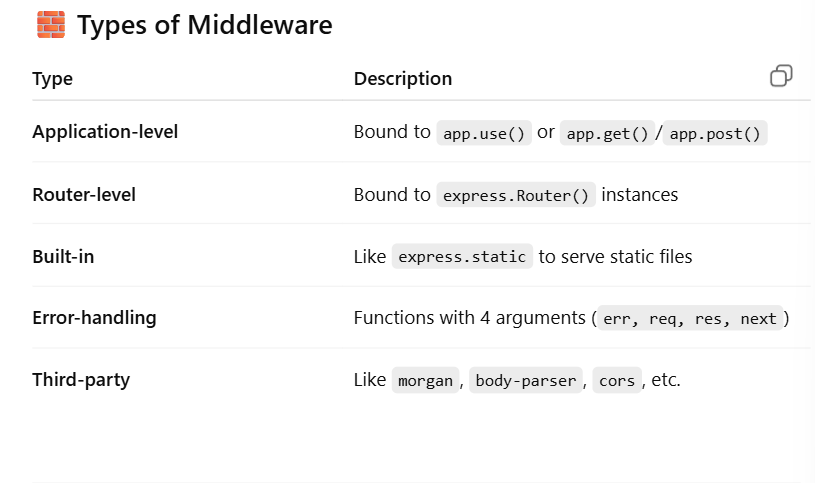
# ✅ Middleware in Express.js

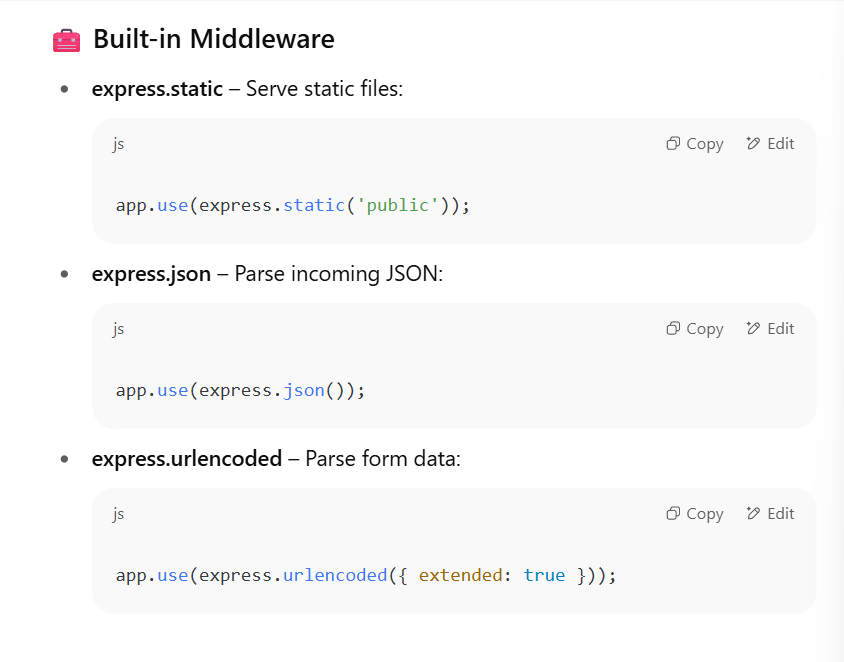
**🔹 What is Middleware?**

In Express.js, **middleware** is a **function** that has access to the request object (req), the response object (res), and the next() function in the application's request-response cycle.

Middleware functions are functions that have access to the request and response objects and can modify them during the lifecycle of a request.



****

**** ****  **🔗 The next() Function**

* Must be called to move to the **next middleware** or **route handler**.
* If not called, the request **hangs** and does not complete.

Imp ponints

🡺 app.use() is the method used to register middleware.

express.static is a built-in middleware to serve static files.

🡺 **A) helmet** – A security-focused third-party middleware, not built into Express by default.

 **B) morgan** – A third-party logging middleware, also not part of Express core.

 **D) nodemon** – Not middleware at all. It’s a development tool used to automatically restart the server on file changes.

🡺You can chain multiple middleware functions using app.get()

🡺cookie-parser is a commonly used third-party middleware.

🡺express.json() parses incoming requests with JSON payloads

express.urlencoded() is used to parse URL-encoded data.

Static files can be served using express.static middleware

Middleware applied using app.use() applies **globally** to all routes.==> **best describes global middleware?**

🡺morgan is a logging middleware commonly used to log HTTP requests.

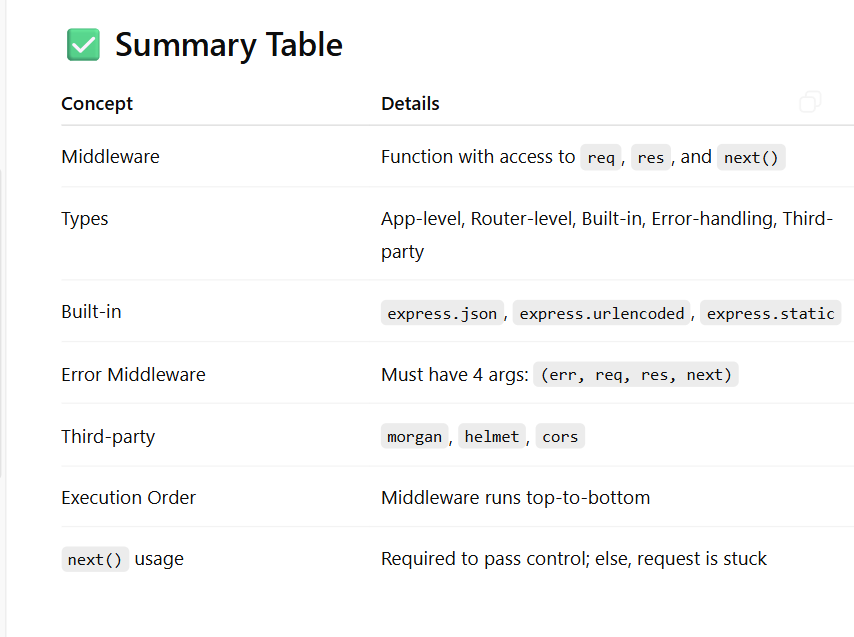
Middleware declared after routes are typically error-handling middleware.

The correct parameter order is (req, res, next).

**Middleware that is applied to a specific route is called:** Route middleware

multer is a middleware used for handling multipart/form-data, used especially for file uploads.

🡺

**middleware is executed in the order it appears.** ****

* **🔍 Concept Summary: Express.js Middleware (MCQs 1–30)**

**1. Middleware Basics:**

* **Middleware functions in Express.js are functions that execute during the request-response cycle.**
* **They receive req, res, and next as arguments.**

**2. Types of Middleware:**

* **Application-level middleware: Added using app.use() or app.METHOD().**
* **Router-level middleware: Scoped to an instance of express.Router().**
* **Built-in middleware: Includes express.json(), express.urlencoded(), express.static().**
* **Third-party middleware: E.g., morgan, cookie-parser, multer.**

**3. Usage and Behavior:**

* **Middleware functions execute in the order they are added.**
* **next() is used to pass control to the next middleware.**
* **If next() is not called, the request will hang.**
* **Returning without next() halts further execution.**
* **Middleware can be synchronous or asynchronous (using async/await).**

**4. Error-Handling Middleware:**

* **Has four arguments: (err, req, res, next).**
* **Should be defined after all other route/middleware definitions.**

**5. Custom Middleware:**

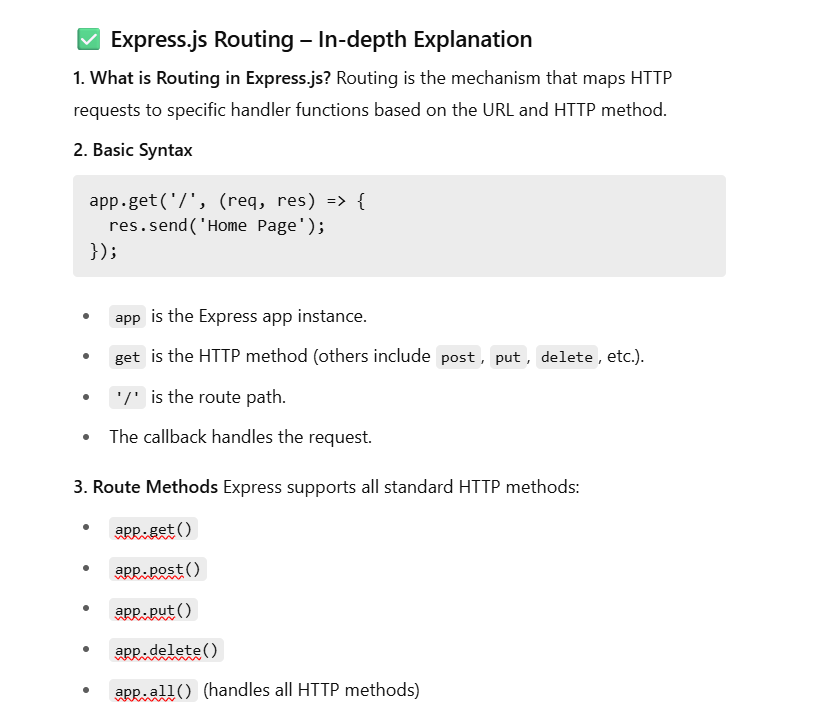
* **Can be created easily with (req, res, next).**
* **Common use cases: logging, authentication, input validation.**

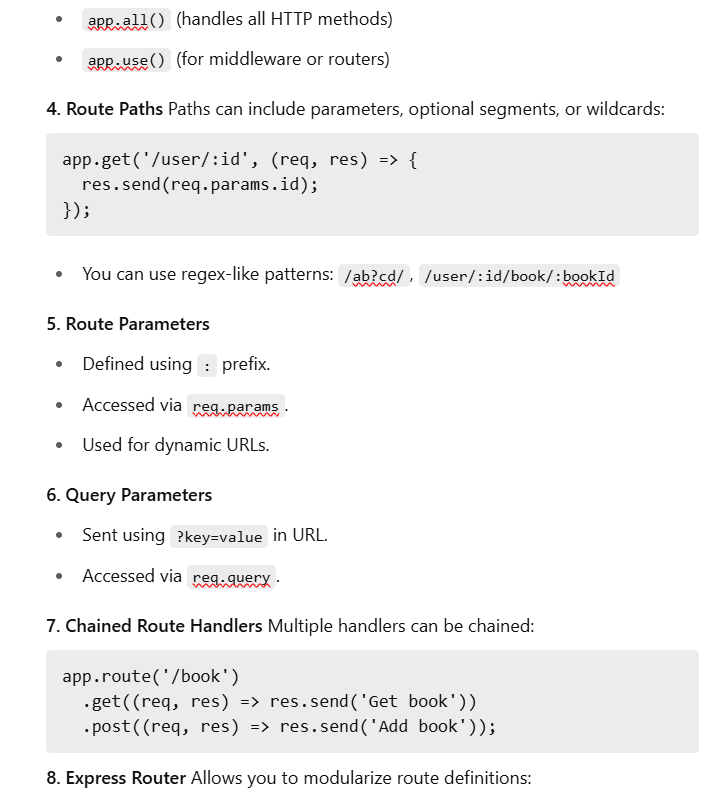
**6. Common Middlewares:**

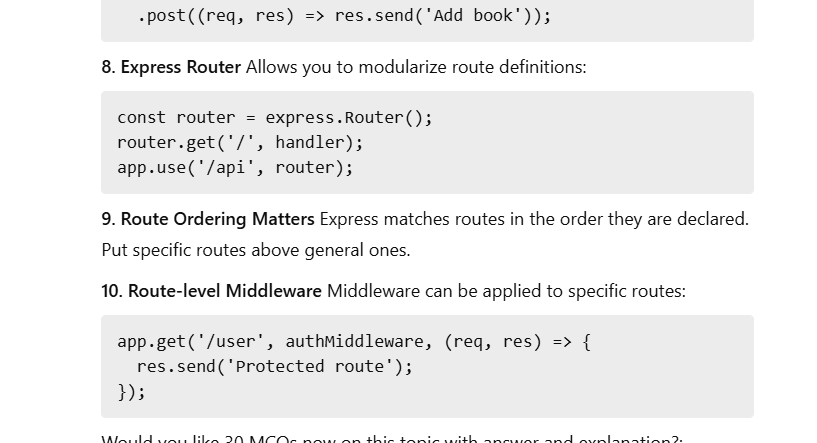
* **morgan: logs HTTP requests.**
* **cookie-parser: parses cookies.**
* **multer: handles file uploads.**
* **express.Router(): defines modular routes and middlewares.**

**7. Execution Flow:**

* **Global middleware runs on all routes.**
* **Route-level middleware is scoped to specific paths.**
* **Multiple middleware can be chained using app.get('/', mw1, mw2, handler).**

****

****

****

**Which of the following is the correct way to handle a click event on a button in React JSX? A) <button onclick="handleClick()">Click Me</button> B) <button onClick={handleClick()}>Click Me</button> C) <button onClick={handleClick}>Click Me</button> D) <button on\_click="handleClick">Click Me</button> Answer: C**

* **Explanation: In React JSX, event handlers are named using camelCase (onClick, onChange, etc.). More importantly, you pass a reference to a JavaScript function (e.g., handleClick) as the value, not a string or the result of a function call (handleClick()). Option B would call handleClick immediately during render, not when the button is clicked.**

** What happens if you define an event handler directly in JSX as an arrow function, like <button onClick={() => console.log('Clicked!')}>? A) The function is called immediately on render. B) A new function is created on every render. C) The button's default click behavior is prevented. D) It will cause an error due to invalid syntax. Answer: B**

* **Explanation: When you define an arrow function inline in JSX, a *new function instance* is created on every re-render of the component. While this is often fine for simple cases, it can sometimes lead to minor performance overhead or issues with memoization in more complex scenarios.**

** In a Class Component, what is the method used to update its internal state? A) updateState() B) changeState() C) setState() D) modifyState() Answer: C**

* **Explanation: Class Components use this.setState() to update their state. This method schedules an update to a component's state object and tells React to re-render the component and its children with the updated state.**

**What is the primary difference in this binding between regular functions and arrow functions when used as event handlers in Class Components? A) Regular functions automatically bind this to the component instance. B) Arrow functions automatically bind this to the component instance. C) Neither automatically binds this; you always need to bind manually. D) this is not a concern in event handlers. Answer: B**

* **Explanation: Arrow functions inherit this from their lexical scope (where they are defined). In a Class Component, an arrow function defined within the class body will have this correctly bound to the component instance. Regular functions do not have this automatic binding and require explicit binding (e.g., this.handleClick = this.handleClick.bind(this); in the constructor).**

**:🡺 useReducer is an alternative to useState for managing more complex state logic, especially when state transitions involve multiple values or when the next state depends on the previous state in a complex way. It's often used with a reducer function, similar to Redux.**

**When should you typically perform data fetching in a Function Component? A) Directly in the component's body before the return statement. B) Inside an onClick handler. C) Inside a useEffect Hook with an appropriate dependency array. D) Within the useState initialization. Answer: C**

* **Explanation: Data fetching is a side effect. It should be performed within a useEffect Hook. An empty dependency array ([]) is common for initial fetches when the component mounts, or with dependencies if the fetch needs to re-run when specific data changes.**

**If you omit the dependency array from useEffect, when will the effect run? A) Only once after the initial mount. B) Only when its parent component re-renders. C) After every render of the component. D) Never. Answer: C**

* **Explanation: If no dependency array is provided, useEffect will run its effect function after every completed render cy**

**What is the purpose of the return statement within the useEffect callback function? A) To return the component's JSX. B) To define a cleanup function. C) To return a new state value. D) To indicate that the effect has completed. Answer: B**

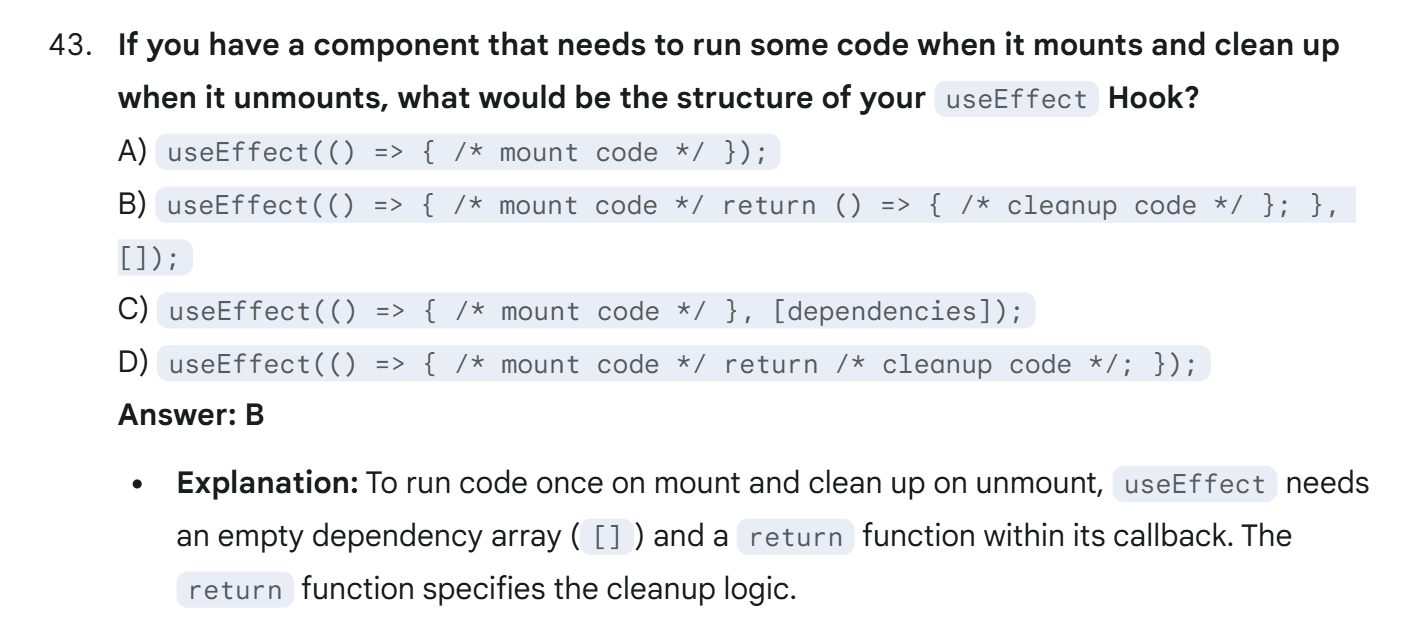
* **Explanation: The function returned by the useEffect callback is the "cleanup" function. React will run this function when the component unmounts or before the effect re-runs (if dependencies change) to clean up resources like subscriptions, timers, or event listeners.**

**componentDidMount is the method in Class Components that runs once after the component's initial render and insertion into the DOM.**

**Which Class Component lifecycle method is called just before a component is unmounted and destroyed? A) componentDidMount B) componentDidUpdate C) componentWillUnmount D) render Answer: C**

* **Explanation: componentWillUnmount is where you'd perform cleanup operations in Class Components, such as invalidating timers or unsubscribing from event listeners, before the component is removed from the DOM.**

**🡺elements like <input>, <textarea>, and <select>, event.target refers to the DOM element that triggered the event, and event.target.value provides its current value.**

****

**React components can only return a single top-level element. React.Fragment (or <></>) allows you to group multiple elements without introducing an unnecessary extra <div> or other element into the actual DOM, which can be useful for styling or semantic HTML.**

**What is the key prop?**

* **A key is a special string attribute you need to include when creating lists of elements.**
* **Purpose: Keys help React identify which items have changed, been added, or been removed. When you have a dynamic list, React uses the key to efficiently update the UI. It helps React's "reconciliation" algorithm.**
* **Importance: Without stable and unique keys, React might re-render the entire list or mismanage elements, leading to performance issues, incorrect state for list items, or unexpected behavior (e.g., text inputs getting mixed up).**
* **Where to put key: The key should be given to the elements inside the map() method, *on the element that is directly returned from the map() callback*. If you're returning a component, the key goes on that component.**

**Choosing a key:**

1. **Unique IDs from your data (Best Practice): If your data items have stable, unique IDs (e.g., from a database), use those.** ****

**What are Refs?**

* **Refs (short for "references") provide a way to access DOM nodes or React elements created in the render method.**
* **They are an "escape hatch" from the declarative React paradigm and should be used sparingly.**

**When to use Refs (Common Use Cases):**

* **Managing focus, text selection, or media playback.**
* **Triggering imperative animations.**
* **Integrating with third-party DOM libraries.**

**Creating and Using Refs (with useRef Hook):**

* **The useRef Hook is used in Function Components to create a ref.**
* **It returns a mutable ref object whose .current property is initialized to the argument passed (null by default).**
* **The .current property will hold the DOM element once it's mounted.**

**Important Notes about Refs:**

* **Don't Overuse: Avoid using refs for anything that can be done declaratively with state and props.**
* **Attached to DOM Nodes: Refs directly interact with the underlying DOM nodes, not necessarily with component instances (unless it's a Class Component ref).**
* **Refs and Function Components: You cannot directly attach a ref to a Function Component itself unless it's wrapped with React.forwardRef. This is an advanced topic.**

### Lifting State Up

This is a fundamental React pattern for sharing state between components that are not directly in a parent-child relationship, or when multiple components need to reflect the same changing data.

**key difference between Link and NavLink components🡺NavLink automatically applies an "active" class or style when its route is matched by the current URL, Link does not**

**The Link component from react-router-dom is indeed the preferred way to create navigation links in React Router that prevent full page reloads and leverage client-side routing.**

**Dynamic segments are denoted by a colon (:) followed by the parameter name, like :id, :productid**

**To replace the current entry in the history stack (preventing the user from going "back" to the previous page), what option can you pass to the Maps function from useNavigate? A) Maps('/new-path', { push: false }); B) Maps('/new-path', { replace: true }); C) Maps('/new-path', { noBack: true }); D) Maps('/new-path', { history: 'replace' }); Answer: B**

* **Explanation: Passing { replace: true } as the second argument to the Maps function tells React Router to replace the current entry in the browser's history stack with the new one, so pressing the back button won't go to the page from which the navigation originated.**

**🡺Outlet🡺It renders the child route's element within a parent route's component (used for nested layouts)**

**🡺useSearchParams is specifically designed to work with URL query parameters. It returns a URLSearchParams object and a setter function, making it easy to read and manipulate query strings. While useLocation provides the raw search string, useSearchParams provides a parsed, convenient interface.**

**What is the significance of the \* wildcard in a Route path (e.g., <Route path="\*" element={<NotFound />} />)? A) It matches any path that contains \*. B) It matches any path starting with \*. C) It acts as a "catch-all" and matches any URL that hasn't been matched by other routes. D) It is used for optional path segments. Answer: C**

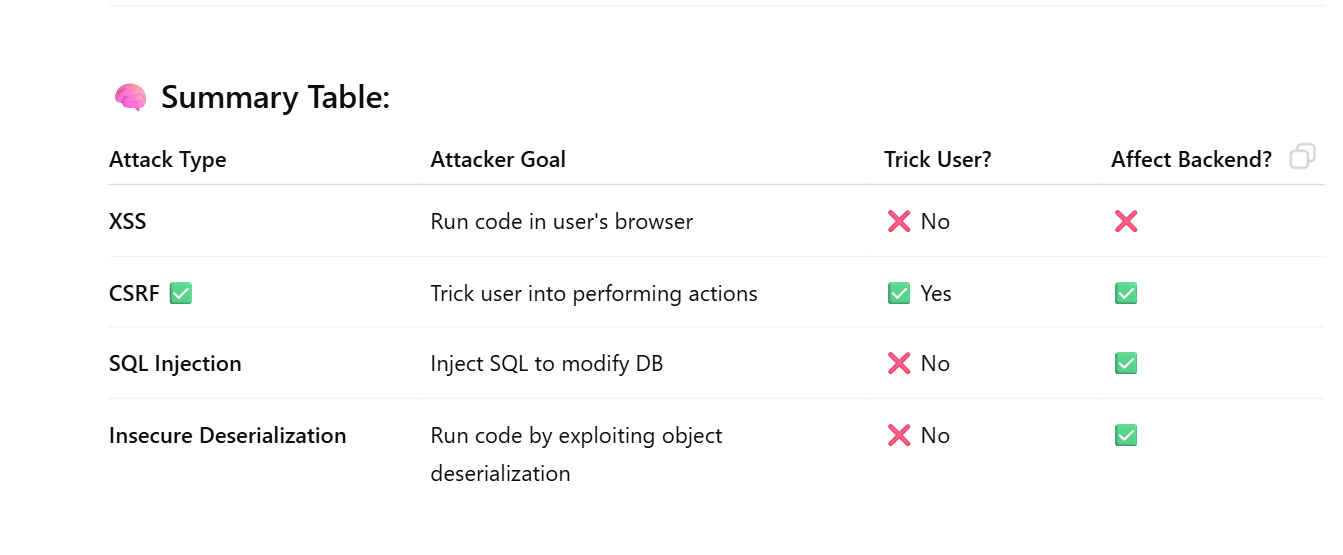
* **Explanation: The \* wildcard is typically placed as the last route in a Routes block. It will match any URL that hasn't been matched by the preceding Route components, making it ideal for a "404 Not Found" page.**

**An index route is rendered when the parent route's path is matched precisely, meaning there are no further sub-paths. It acts as the default content for the parent segment.**

**document.querySelectorAll(selector):**

* **Returns a NodeList (a static, array-like object) of *all* elements that match the specified CSS selector.**

**NodeList *does* support forEach directly.**

**getElementsByTagName returns an HTMLCollection. You cannot apply .style directly to the collection. You must loop through its elements. forEach() method: Directly available on NodeList (from querySelectorAll), but not directly on HTMLCollection. You can convert an HTMLCollection to an array using Array.from() to use forEach.** ****

[**https://g.co/gemini/share/9191f61e210b**](https://g.co/gemini/share/9191f61e210b) **gemni**

[**+**](https://chatgpt.com/share/686d4db8-03f0-800d-9170-a230a059e020) **chatgpt**

**https://chatgpt.com/share/686d4e27-4d64-800d-a458-c73cc3b30c3b**

**Notes**

HTTP (Hypertext Transfer Protocol): Protocol for web communication🡺Works on port 80

**c) HTTP Status Codes:**

* **1xx:** Informational
* **2xx:** Success (e.g., 200 OK)
* **3xx:** Redirect (e.g., 301 Moved)
* **4xx:** Client error (e.g., 404 Not Found)
* **5xx:** Server error (e.g., 500 Internal Server Error)

**e) HTTPS: 🡺**Works on **port 443**

**b) HTTP Methods:**

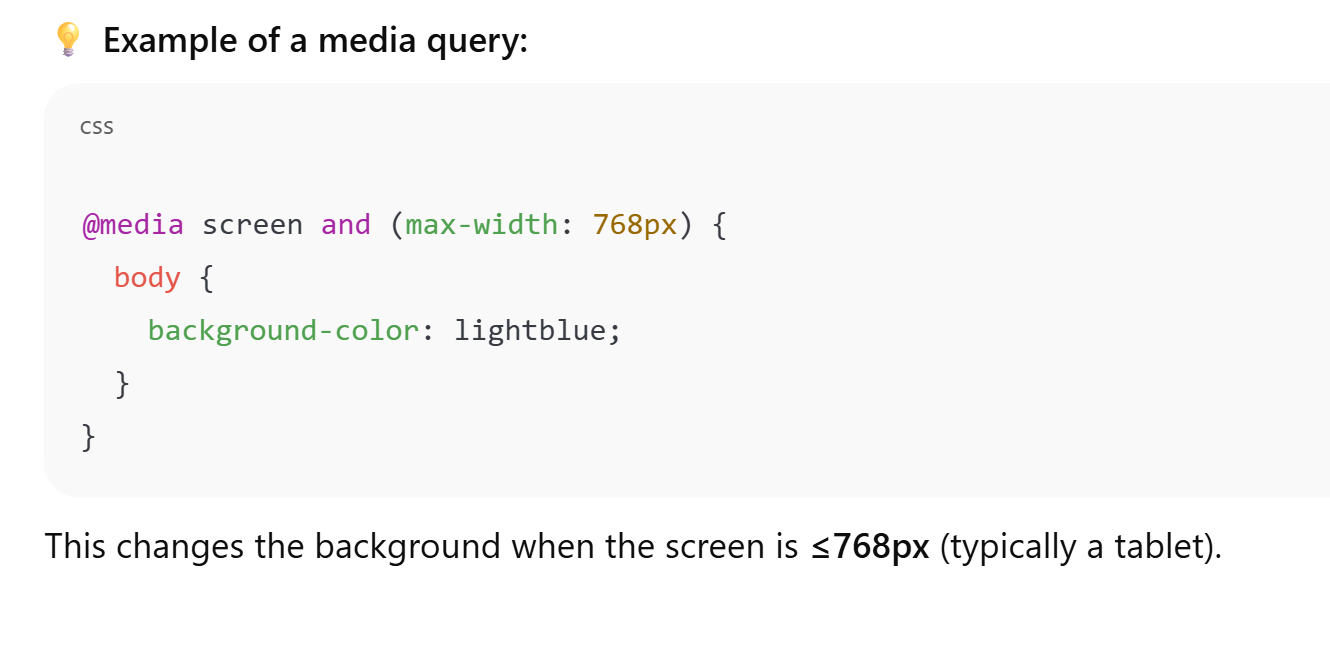
* **GET** – retrieve data
* **POST** – submit data
* **HEAD** – headers only, no body
* **PUT** – update data
* **DELETE** – remove resource

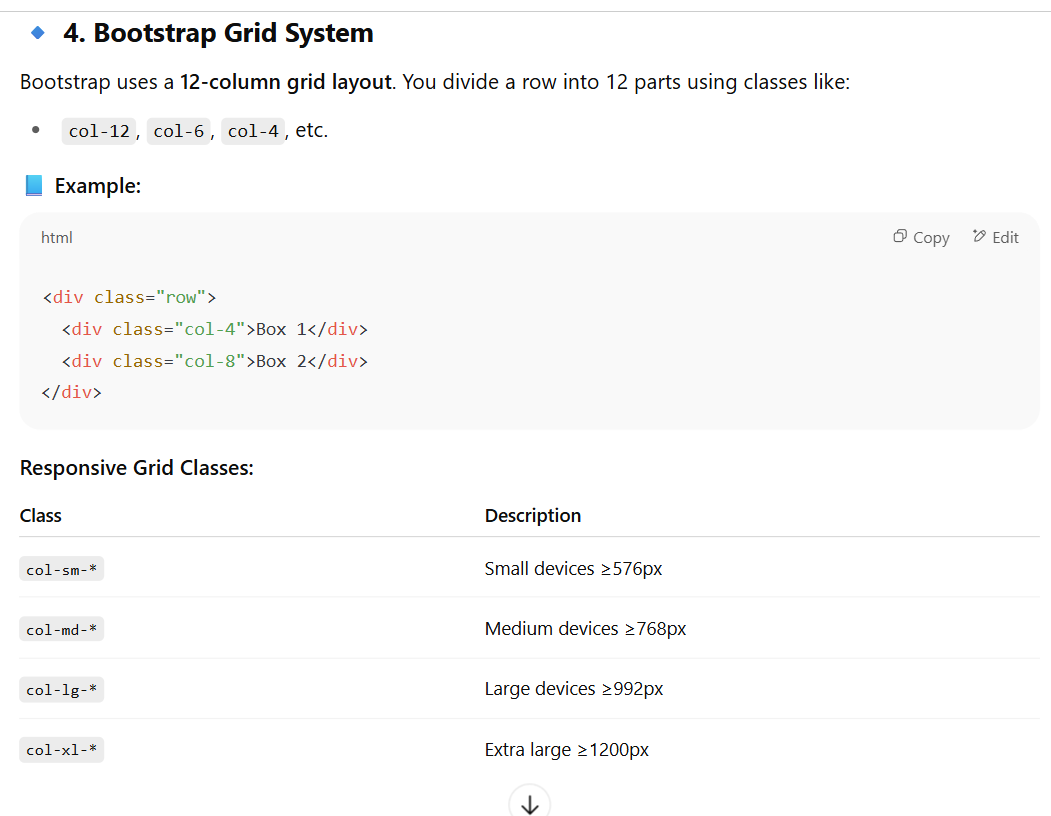
# ✅ Session 4: Responsive Web Design

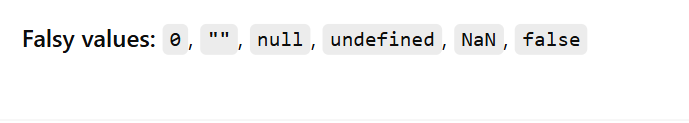
**1. What is Responsive Web Design (RWD)?**

**📘 Key Techniques:**

* **Fluid Grids**
* **Flexible Images**
* **Media Queries**

****

****

****