DAY 20 & 21 topics

Advanced Java Script	Modern JavaScript (ES6): Javascript - Fetch API , Arrow functions, Template literals, Destructuring, Spread and rest operators
	Date and Time Objects, Closures, Promises and Async Programming, Closures and Scoping,
	DOM Manipulation: DOM Objects, Accessing elements in the DOM, Modifying HTML content, Adding and removing elements
	Functional Programming, Error Handling client side scripting
	JSON, JSON - Objects, arrays etc, JSON-Ajax Advanced DOM Manipulation,
	jQuery Plugins, Event Delegation, Ajax and Deferred Objects, Optimization and Performance Diff between JQuery and Bootstrap ? AJAX - Asynchronous javascript and XML

How can you say Js supports modern solution building ?

1. It supports Asynchronous programming (Multithreading)
2. It can handle real time data(API)
3. Write clean, reusable code
4. Build responsive and interactive UI
5. Optimize performance and User experience.

How can you say JavaScript is ideal for Enterprise application building ?

1. Scalability
2. Maintainability
3. Security
4. High performance

Year / Era	HTML Evolution	CSS Evolution	JavaScript Evolution
1991–1993	HTML 1.0 created by Tim Berners-Lee for sharing documents	Not available	Not available
1995	HTML 2.0 (basic forms, links)	Inline styling via HTML attributes	JavaScript created by Brendan Eich (Netscape) in 10 days
1996–1997	HTML 3.2 (tables, layout hacks)	CSS 1 introduced (separation of style)	JavaScript standardized as ECMAScript (ES1)
1999	HTML 4.01 (semantic tags, frames)	CSS 2 (positioning, z-index)	ES3 (loops, regex, error handling)
2000–2005	XHTML attempts strict XML-based HTML	Limited adoption of CSS 2	JavaScript used mainly for form validation
2006–2009	HTML stagnation	CSS 2.1 stabilizes	AJAX popularized (Google Maps, Gmail)
2010	HTML5 announced (audio, video, canvas)	CSS3 modules (animations, transitions)	JavaScript becomes critical for UI interactions
2015	HTML5 finalized	CSS Flexbox & media queries mature	ES6 released (arrow functions, classes, promises)
2016–2019	Semantic HTML improves accessibility	CSS Grid introduced	JavaScript frameworks dominate (Angular, React, Vue)
2020–Present	HTML stable, incremental updates	Modern CSS (variables, container queries)	JavaScript everywhere (frontend, backend, mobile, AI)

Feature / Aspect	JavaScript (ES5 – Before 2015)	ES6 (ECMAScript 2015 & Later)
Release Year	2009 (ES5)	2015 (ES6)
Variable Declaration	`var` only	`let`, `const`
Scope	Function scope	Block scope (`let`, `const`)
Constants	Not supported	`const` keyword
Arrow Functions	❌ Not available	✅ `() => {}`
Function Syntax	Verbose	Short and cleaner
`this` Keyword	Dynamic, confusing	Lexical binding in arrow functions
String Handling	Concatenation using `+`	Template literals `` `${}` ``
Multi-line Strings	❌ Not supported	✅ Supported
Default Parameters	❌ Not supported	✅ Supported
Destructuring	❌ Not available	✅ Array & Object destructuring
Spread Operator	❌ Not available	✅ `...`
Rest Parameters	❌ Not available	✅ `...args`
Classes	Prototype-based only	`class` syntax introduced
Inheritance	Complex prototype chaining	Simple `extends` keyword
Modules	❌ No native modules	✅ `import` / `export`
Promises	Limited support	Native Promise support
Async Programming	Callbacks	Promises, Async/Await (later ES)

Loops	`for`, `while`	`for...of` added
Object Creation	Verbose syntax	Shorthand property & method syntax
Parameter Handling	Manual checks	Cleaner default params
Code Readability	Lower	Higher
Maintainability	Harder in large apps	Easier for large-scale apps
Browser Support	Very high	Modern browsers (transpilers help)

Arrow functions (=>)

1. It is shorter, cleaner syntax for writing functions in javascript.
2. It reduces boilerplate code.
3. Fix **this** binding issues.(How this keyword behaves)
4. Improves readability in callbacks and functional programming.

Syntax:

- basic : (a,b) => a+b
- Single parameters : x => x*2
- No parameter : ( ) => console.log("hi")
- Multiple statements : (a,b) => { return a+ b; }
- Returning object : () => ({ name:"JS"})

function Timer() {
this.seconds = 0;
setInterval(function () {
this.seconds++;
}, 1000);
}
 	
Here this refers to the window
	function Timer() {
	this.seconds = 0;
	setInterval(() => {
	this.seconds++;
	}, 1000);
	}
	
	this refers to the timer

When not to use arrow function:

1. Object methods : "this" keyword may break
2. Event handlers : Need Dynamic this
3. Constructors : Not allowed

Some common real world applications :

1. Arrays methods(map, filter, reduce)
2. Events Callbacks
3. Promise handling
4. React functional components

"Arrow functions simplify syntax and inherit `this` from their surrounding scope. "

10-Minute Case Study: Arrow Functions in Action

Case Study Title

"QuickCart – Lightweight Product Utility Module"

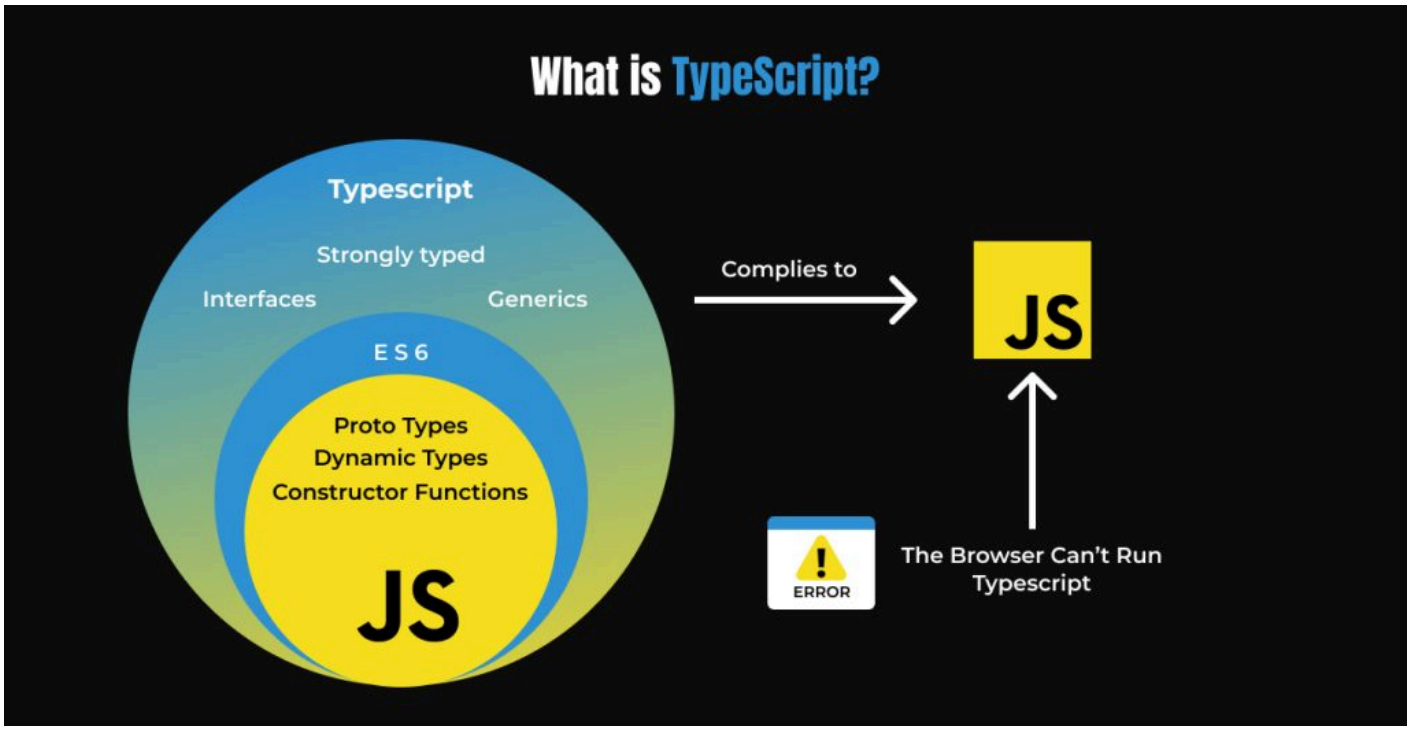
Context

QuickCart is a simple client-side utility used in an e-commerce application to:

- Display messages
- Perform price calculations
- Process product lists
- Handle object creation

Story ID	User Story	Arrow Function Type Used	Sample Implementation

US-01	As a user, I want to see a welcome message when the app loads so that I know the system is ready.	No-parameter arrow function	`const showWelcome = () => console.log("Welcome");`
US-02	As a shopper, I want to calculate discounted price so that I know how much I save.	Single-parameter arrow function	`const applyDiscount = price => price * 0.9;`
US-03	As a customer, I want my total bill calculated with tax so that I know final payable amount.	Multiple-parameter arrow function	`const calculateTotal = (price, tax) => price + tax;`
US-04	As a system, I want to generate billing summary so that pricing details are clear.	Arrow function with block body	`const summary = (p,d)=>{return {final:p-d};};`
US-05	As a developer, I want to create product objects quickly so that code remains clean.	Arrow function returning object	`const createProduct = (id,name)=>({id,name});`
US-06	As a user, I want discounted prices for all products so that I can compare options.	Arrow function as callback (`map`)	`prices.map(p => p * 0.9);`
US-07	As a system, I want to track session time correctly so that activity is monitored.	Arrow function with lexical `this`	`setInterval(()=>{this.sec++;},1000);`



Fetch API :- (reading some data from API) (**JS functionality provide by browser to call API over HTTP)**
The fetch APi is a modern , promise-based way to make Http requests from browsers, replacing older approaches like XMLHttpRequest.

What all problems it solves : -

1. Callback hell (v.v.v imp):
2. Complex AJAX syntax
3. Poor error handling

Syntax : fetch(url) .then(response => response.json()) .then(data =>console.log(data)) .catch(error=> console.log(error));	**Fetch Workflow:** Step 1: Browser sends HTTP request Step 2: Server responds Step 3: response object returned Step 4: Data is parsed(JSON/Txt) Step 5: UI is updated

What is call back hell : (very common in old AJAX based on XMLHTTPRequest)

Call back hell happens when **multiple asynchronous operations** are nested inside each other, making the code:

1. Hard to read
2. Hard to debug
3. Hard to maintain

Ex. You want to order food online. Steps: 1)Check restaurant availability 2)Check the menu 3)place order 4)Make payment 5)Track Delivery All above logic is nested(as each step is dependent on previous step)	

1. Deep nesting
2. Difficult to read
3. Error handling is scattered
	getResturant(function(resturant) {
	getMenu(restaurant, function(menu) {
	PlaceOrder(menu, function (order) {
	makePayment(order, function(payment) {
	Trackdelivery(payment, function (status) {
	console.log(status);
	});
	});
	});
	});
	});

We can solve above issue using fetch(URL)	fetchResturant() .then(getMenu) .then(Placeorder) .then(MakePayment) .then(TrackDEliver) .then(status => console.log(status)) .catch(error=> console.log(error));

Problem	Before Fetch	After Fetch
Callback Hell	Nested callbacks	Promises / Async-Await
AJAX Syntax	Verbose & complex	Simple & readable
Error Handling	Manual & scattered	Centralized & clean

Maintainability	Poor	High
Readability	Low	High

What is API ?
Application programming interface
It is a contract that allows two software systems to communicate with each other in a predefined way.

request => User(Your application) -> order -> Waiter (Acting as API)-> Kitchen DB)/Server-> Chef
response =>Chef -> waiter(API) -> User table

“ Fetch API is how Js talks to APIs”

API v/s REST API v/s Graph API v/s SOAP API
Diff between fetch API and AXios in JS

Create a Web application to fetch API given below and display data on browser

```JavaScript
https://jsonplaceholder.typicode.com/todos/
https://jsonplaceholder.typicode.com/comments
https://jsonplaceholder.typicode.com/users
```

Template literals (`) (Instead of using + for concatenating two strings, we can use $

- Primarily used to work with string
- We can embed expression into string , multiline strings
- String interpolation (Common for most of the front end frameworks)
- Multi line strings
- Expressions evaluation
- Cleaner concatenation

Syntax: `string text ${expression} string text`

real world use cases:

1. Dynamic mail template
2. UI messages
3. Logging and debugging
4. API response formatting.

best practices :

1. Prefer keeping literal over strings concatenation
2. Keep expressions as simple as possible. ${ }
3. Avoid heavy logic inside templates.

Common pitfalls :

1. Using single or double quotes instead of backticks
    a. ex He said that " I am good in programming " ;
2. Overusing nested expressions

Destructuring : Unpacking of values from arrays or objects into variables.

Arrays DEstructuring :

const[a,b] = array;

Objects(a,b,c,)

real world use cases :

1. API response handling.
2. Function parameter extraction
3. Cleaner react Props usage
4. Configuration parsing.

Best practices :

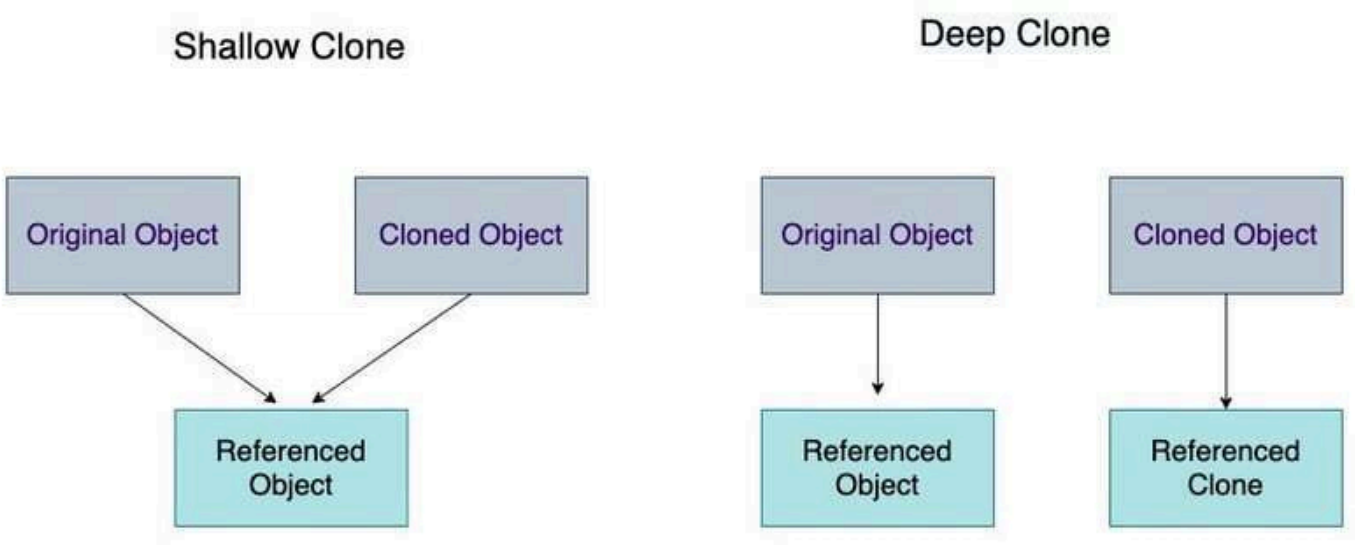
1. Destructure only required fields.
2. Use defaults to avoid undefined errors.
3. Avoid deep destructuring when readability suffers.

Spread and rest operators

Spread Operator (. . .)	Rest Operator (. . .)
Expands values	**Collects values**
Used while creating arrays, objects, or function	**Used while receiving function parameters or**

calls	destructuring
Many → Individual	Individual → Many
Used on the right-hand side	Used on the left-hand side
Creates shallow copies (creating a new references for the outer object)	Gathers remaining elements
Example: `[...arr1, ...arr2]`	Example: `function fn(...args)`
Example: `{ ...obj1, ...obj2 }`	Example: `const { a, ...rest } = obj`
Example: `fn(...values)`	Example: `const [x, ...others] = arr`
Common in React for state updates	Common in APIs with variable arguments
Cannot be used in function parameter list	Must be the last parameter

Deep copy and shallow copy : Mysterious FAQ



Spread	Rest
Expands values	Collects values

Used while creating	Used while receiving
Many → Individual	Individual → Many

Date and time : **In JS we have built-in Date object to work with:**

1. **Date**
2. **Time**
3. **Timezones**
4. **Timestamps**

Note: Internally JS stores dates as the number of milliseconds since 1 jan 1970(UTC) called as Unix Epoch

Method	Description
`getFullYear()`	Year (YYYY)
`getMonth()`	Month (0–11)
`getDate()`	Day of month
`getDay()`	Day of week (0–6)
`getHours()`	Hours
`getMinutes()`	Minutes
`getSeconds()`	Seconds
`getTime()`	Timestamp (ms)

Scenario	Usage
Logging	**Timestamps**
Scheduling	**Session start/end**
Deadlines	**Due date comparison**
UI display	**Locale formatting**
APIs	**ISO strings**