Day 02: Advanced JavaScript Refresher

📌 ES6+ Features & Core Concepts for React

Sharpen your JavaScript skills for React development.

Agenda

01

ES6+ Essentials

Key modern JavaScript features.

02

Objects & Arrays

Advanced handling and manipulation.

03

Array Methods

Powerful transformations and iterations.

04

Asynchronous JavaScript

Handling non-blocking operations.

05

Modules (import/export)

Organising your codebase.

06

Closures & Scopes

Deep dive into JavaScript's core.

ES6+ Essentials: `let` & `const`

`let` → Block scoped, reassignable

`const` → Block scoped, not reassignable

```
let count = 1;
count = 2; //
```

ES6+ Essentials: Arrow Functions

Arrow functions provide a more concise syntax for writing function expressions. They also handle the `this` keyword differently, binding it lexically to the surrounding scope, which is crucial for managing context in React components.

- Shorter syntax
- `this` is lexically scoped

```
// Traditional function
function greet(name) {
 return "Hello, " + name;
}

// Arrow function
const greet = (name) => `Hello, ${name}!`;
```

ES6+ Essentials: Template Literals

Template literals, enclosed by backticks (\`), offer a powerful way to embed expressions within strings using ${expression}. This makes string concatenation cleaner and more readable, especially when dealing with dynamic content in your UI.

- Embed expressions with backticks \`
- Multi-line string support

```
const name = "Hashan";
const msg = `Welcome, ${name}!
```

Spread & Rest Operators

Spread (...): Expands iterables (like arrays or strings) into individual elements. It's great for combining arrays or creating shallow copies of objects.

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray]; // [1, 2, 3]

const originalObject = { a: 1, b: 2 };
const copiedObject = { ...originalObject }; // { a: 1, b: 2 }
```

Rest (...): Collects multiple elements into a single array. Used in function parameters to handle an indefinite number of arguments.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3)); // Output: 6
console.log(sum(10, 20, 30, 40)); // Output: 100
```

Destructuring

Destructuring assignment is a powerful ES6+ feature that allows you to unpack values from arrays or properties from objects into distinct variables. This simplifies code and makes it more readable, especially when dealing with props in React components.

```
const user = { name: "Janith", age: 25 };
const { name, age } = user; // name: "Janith", age: 25

const [a, b] = [10, 20]; // a = 10, b = 20

const numbers = [1, 2, 3, 4, 5];
const [first, second, ...rest] = numbers; // first=1, second=2, rest=[3,4,5]
```

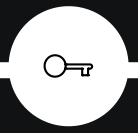
This technique significantly cleans up data extraction, reducing boilerplate and improving code clarity.

Objects & Arrays: Enhancements

Shorthand Properties

When a property name matches a variable name, you can omit the value.

```
const name = "React";
const obj = { name }; // { name: "React" }
```



Computed Properties

Allows you to use an expression for a property name, enclosed in square brackets.

```
const key = "id";
const obj = { [key]: 123 }; // { id: 123 }
```

These features streamline object creation, especially when dynamically generating objects or mapping data structures.

Array Methods: map, filter, reduce

These higher-order array methods are essential for functional programming patterns in JavaScript and are heavily used in React for data manipulation.

map()

→ transform each element into a new array.

filter()

→ create a new array with elements that pass a test.

reduce()

→ accumulate values into a single result.

```
const nums = [1, 2, 3, 4];
nums.map(x => x * 2); // [2,4,6,8]
nums.filter(x => x % 2); // [1,3]
nums.reduce((a, b) => a + b); // 10
```

Array Methods: `find` & Others

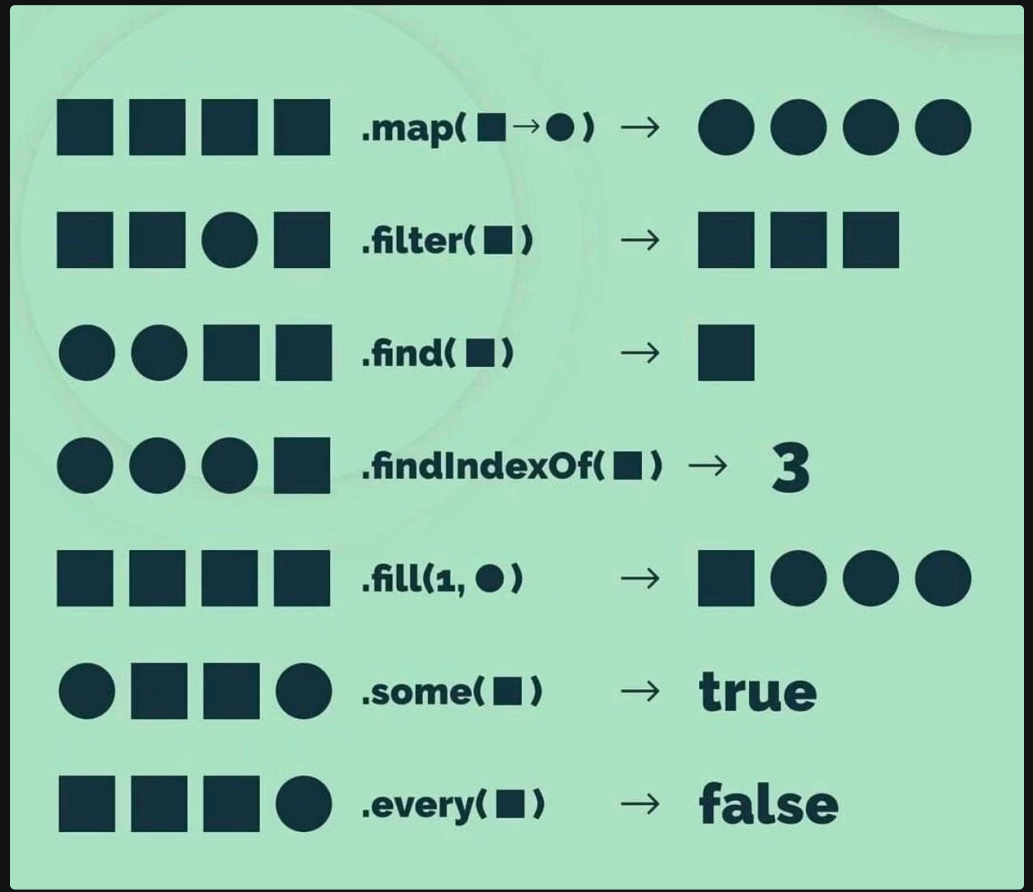
`find()`

Returns the **first element** in the array that satisfies the provided testing function. Otherwise, it returns `undefined`.

```
const users = [{id:1}, {id:2}];
users.find(u => u.id === 2); // {id:2}
```

Other useful methods:

- `some()`**:** Checks if at least one element satisfies the condition.
- `every()`**:** Checks if all elements satisfy the condition.
- `includes()`**:** Checks if an array contains a certain value.



These methods provide efficient ways to query and inspect array content, improving code readability and performance.

```
/  JavaScript Array Methods

[3, 4, 5, 6].at(1); // 4
[3, 4, 5, 6].pop(); // [3, 4, 5]
[3, 4, 5, 6].push(7); // [3, 4, 5, 6, 7]
[3, 4, 5, 6].fill(1); // [1, 1, 1, 1]
[3, 4, 5, 6].join("-"); // '3-4-5-6'
[3, 4, 5, 6].shift(); // [4, 5, 6]
[3, 4, 5, 6].reverse(); // [6, 5, 4, 3]
[3, 4, 5, 6].unshift(1); // [1, 3, 4, 5, 6]
[3, 4, 5, 6].includes(5); // true
[3, 4, 5, 6].map((num) => num + 6); // [9, 10, 11, 12]
[3, 4, 5, 6].find((num) => num > 4); // 5
[3, 4, 5, 6].filter((num) => num > 4); // [5, 6]
[3, 4, 5, 6].every((num) => num > 5); // false
[3, 4, 5, 6].findIndex((num) => num > 4); // 2
[3, 4, 5, 6].reduce((acc, num) => acc + num, 0); // 18
```

Asynchronous JavaScript

Promises: The Foundation of Async

JavaScript's non-blocking nature is crucial for smooth user experiences. Promises provide a structured way to handle asynchronous operations, representing the eventual completion (or failure) of an async task.

```
const fetchData = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data loaded"), 1000);
  });
};

fetchData().then(data => console.log(data));
```

This pattern elegantly manages pending states, successful outcomes, and errors, ensuring code readability and maintainability.

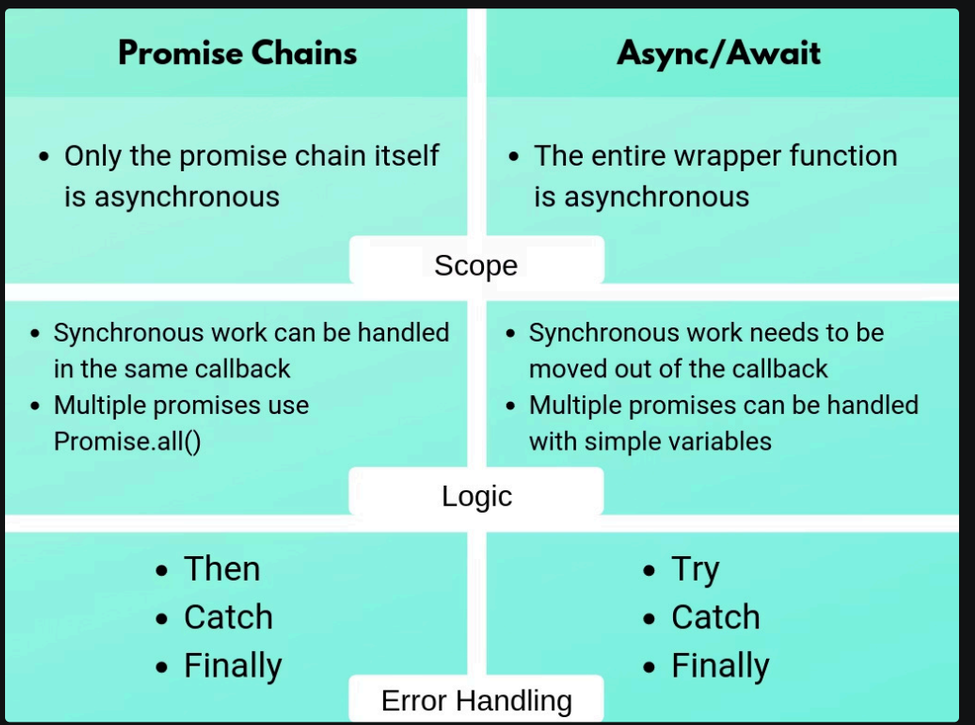
Streamlining Asynchronicity

Async/Await: A Cleaner Promise Syntax

While Promises are powerful, nested `.then()` calls can lead to 'callback hell'. `async/await` offers a more synchronous-looking way to write asynchronous code, making it significantly more readable and easier to debug.

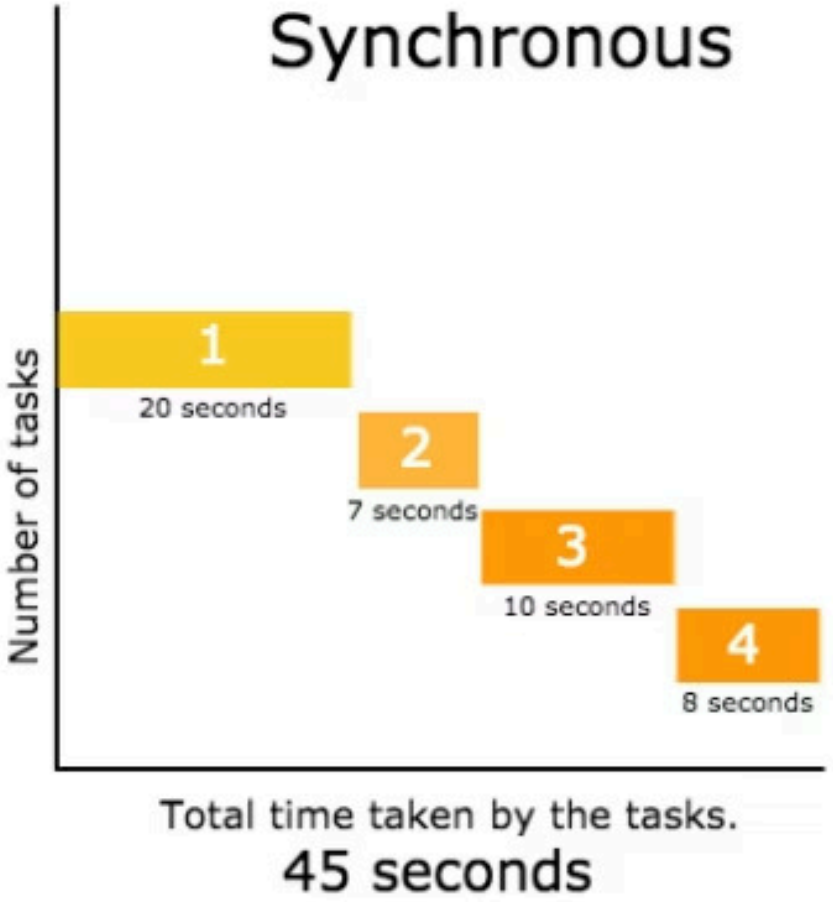
```
async function loadData() {
  const data = await fetchData();
  console.log(data);
}

loadData();
```



This syntax pauses execution until the Promise resolves, just like a synchronous function, but without blocking the main thread.

Synchronous vs Asynchronous



Data Fetching Made Easy

Fetch API: Modern Web Requests

The Fetch API provides a powerful and flexible interface for making network requests, replacing older methods like `XMLHttpRequest`. It leverages Promises, making it perfectly compatible with `async/await` for clean, readable code.

```
async function getUsers() {
  const res = await fetch("https://jsonplaceholder.typicode.com/users");
  const users = await res.json();
  console.log(users);
}

getUsers();
```

It's the standard for modern web development, handling everything from simple GET requests to complex data submissions.

Organizing Your Codebase

Modules: Import & Export

JavaScript modules allow you to break your code into smaller, reusable pieces, improving maintainability, reusability, and preventing global namespace pollution. They promote a modular architecture, essential for larger applications.

1

Named Exports

Allows exporting multiple values from a module. You import them by their specific names.

```
// utils.js
export const add = (a,b) => a+b;
```

```
// main.js
import { add } from "./utils.js";
```

2

Default Exports

Allows exporting a single value as the module's primary export. You can name it anything on import.

```
// greet.js
export default function greet() { /* ... */ }
```

```
// main.js
import greet from "./greet.js";
```

Understanding Execution Contexts

Closures & Lexical Scope

Closures are a fundamental concept where a function "remembers" its surrounding environment (its lexical scope) even after that environment has finished executing. This allows inner functions to access variables from their outer functions.

```
function counter() {
  let count = 0;
  return () => ++count;
}

const inc = counter();
inc(); // 1
inc(); // 2
```

Lexical Scope: A function's scope is determined by where it is defined, not where it is called.

Scope

Global Scope

```
let year = '2020';
```

Function Scope

```
function theYear() {
    let text = "The year is"
    return text + " " + year;
}
```

Block Scope

```
if(10 < 20) {
    let greeting = "hi";
    return greeting
}
```

Closures in Action

React Hooks & Closures

Closures are integral to how React Hooks (like `useState` and `useEffect`) manage state and side effects. Each time a component renders, its functions form closures over the current props and state, ensuring they always have access to the correct values.

```
function Counter() {
  const [count, setCount] = React.useState(0);

  function increment() {
    setCount(c => c + 1); // closure over "c"
  }

  return {count};
}
```

This mechanism is why functional components can maintain their state across re-renders without explicitly binding `this`, simplifying component logic.