📘 Chapter 0.2

JavaScript for React — Every Concept You MUST Know (Nothing Skipped)

> This chapter answers one question completely:
> **“What JavaScript knowledge does React strictly depend on?”**

Not general JS.
Not interview JS.
React-critical JavaScript.

1️⃣ EXECUTION MODEL — HOW JAVASCRIPT RUNS (VERY IMPORTANT)

Why This Matters for React

React code runs **top to bottom, again and again** on re-renders.
If you don’t understand how JS executes, React will feel random.

JavaScript Is:

- Single-threaded
- Synchronous by default
- Event-driven

This means:

- One thing runs at a time
- Long tasks block UI
- Async work must be handled carefully

Example

```
console.log("A");

setTimeout(() => {
  console.log("B");
}, 0);

console.log("C");
```

Output:

```
A
C
B
```

Why this matters in React:
Effects, events, and state updates follow this exact model.

2️⃣ VARIABLES — `var`, `let`, `const` (STRICT RULES)

Why React HATES `var`

- Function scoped
- Hoisted unpredictably
- Causes UI bugs

React rule:

❌ Never use `var`
✅ Use `const` by default
✅ Use `let` only when reassignment is unavoidable

Example (React-relevant)

```
const status = "SAFE";
// status = "DANGER" ❌ not allowed
```

This prevents accidental mutation.

3️⃣ DATA TYPES — WHAT REACT ACTUALLY USES

React mainly works with:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`
- `array`
- `function`

Truthy / Falsy (CRITICAL)

```
false
0
""
null
undefined
NaN
```

Why this matters:

```
{isActive && <Alert />}
```

Falsy values **control rendering.**

4️⃣ FUNCTIONS — CORE OF REACT

Function Declaration vs Expression

```
function fn() {}
const fn = () => {}
```

React prefers **arrow functions** because:

- No `this`
- Predictable scope
- Cleaner callbacks

Functions as Values (VERY IMPORTANT)

```
function handleSOS() {}
<button onClick={handleSOS} />
```

In React:

- Functions are passed
- Stored
- Reused
- Memorized

This is why `useCallback` exists later.

5️⃣ OBJECTS — STATE & PROPS LIVE HERE

Object Basics

```
const user = {
  name: "Amit",
  active: true
};
```

Object Reference (CRITICAL)

```
const a = {};
const b = a;

b.x = 1;
console.log(a.x); // 1
```

Why React cares:
React checks **reference**, not content.

6️⃣ IMMUTABILITY — THE MOST IMPORTANT RULE

❌ Wrong (Mutation)

```
user.location = "Delhi";
setUser(user);
```

✅ Correct (Immutable update)

```
setUser(prev => ({
  ...prev,
  location: "Delhi"
}));
```

Why:

- New reference
- React detects change
- UI updates safely

7️⃣ ARRAYS — LIST RENDERING & STATE

Array Copying

```
const arr = [1, 2, 3];
const newArr = [...arr, 4];
```

Array Methods React Uses

`map`

```
alerts.map(a => <Alert key={a.id} />)
```

`filter`

```
alerts.filter(a => a.active)
```

`reduce`

Used for:

- Counts
- Aggregates
- Derived state

8️⃣ DESTRUCTURING — CLEAN COMPONENTS

Objects

```
const { name, status } = props;
```

Arrays

```
const [state, setState] = useState();
```

This syntax **is destructuring**.

Without this, hooks make no sense.

9️⃣ SPREAD OPERATOR (`...`) — HOW REACT UPDATES STATE

Objects

```
{ ...prev, active: true }
```

Arrays

```
[...prev, newItem]
```

Spread:

- Copies
- Prevents mutation
- Enables re-render

🔟 CONDITIONAL LOGIC — HOW UI DECIDES

Ternary

```
{isEmergency ? <SOS /> : <Safe />}
```

Logical AND

```
{loading && <Spinner />}
```

React does **not** allow:

```
if (x) { ... } // ❌ inside JSX
```

This is JS expression vs statement knowledge.

1️⃣ 1️⃣ MODULES — HOW REACT SCALES

Why Modules Exist

Large apps need:

- Separation
- Isolation
- Reuse

Export

```
export default Component;
export const helper = () => {};
```

Import

```
import Component from "./Component";
import { helper } from "./utils";
```

React architecture **depends on modules**.

1️⃣ 2️⃣ EVENTS — HOW USERS TALK TO UI

Event Object

```
<input onChange={e => setValue(e.target.value)} />
```

Events are:

- Synthetic (React wraps native events)
- Pooled (older versions)
- Predictable

Understanding `e.target.value` is mandatory.

1️⃣ 3️⃣ ASYNC JAVASCRIPT — TIME & NETWORK

Promise Basics

```
fetch(url).then(res => res.json());
```

`async / await`

```
async function load() {
  const data = await fetchData();
}
```

React uses this with:

- `useEffect`
- API calls

- Side effects

1️⃣4️⃣ CLOSURES — SILENT REACT BUG SOURCE

Example

```
function Counter() {
  let count = 0;

  function increment() {
    count++;
  }
}
```

In React:

- Closures can capture stale values
- Dependency arrays exist to fix this

This concept is **critical for hooks**.

1️⃣5️⃣ SHORT-CIRCUIT & DEFAULTS

```
const name = user.name || "Guest";
```

Used everywhere in UI rendering.

1️⃣6️⃣ OPTIONAL CHAINING

```
user?.profile?.name
```

Prevents UI crashes.

Mandatory for safe rendering.

1️⃣ 7️⃣ WHY THIS ALL MATTERS FOR REACT

Every React concept:

- Hooks
- State
- Effects
- Context
- Performance

Is built on this JavaScript foundation.

Weak JS → broken React
Strong JS → calm React

🛠️ MINI PROJECT (NON-NEGOTIABLE)

Project: JavaScript Core for React

Folder:

```
chapter-0-2-js-complete/
```

Tasks:

1. Create immutable user state updates
2. Render mock UI data using `map`
3. Write async fetch simulation

4. Demonstrate mutation bug vs immutable fix
5. Use destructuring everywhere

Document:

- What breaks
- Why React needs each concept

🧠 FINAL THOUGHT

React is not a framework that hides JavaScript.
React is a **discipline that exposes weak JavaScript**.

You are now building a **real foundation**, not rushing tutorials.