Session 1

The JavaScript Runtime

Memory Management: Stack vs. Heap

Understanding `const` requires understanding memory assignment.

- **Primitives (Stack):** Numbers, Booleans, undefined.
- **References (Heap):** Objects, Arrays, Functions.

Crucial Concept: `const` does not create an immutable value; it creates an immutable **binding**.

```
const user = { name: "Alice" };
user.name = "Bob"; // Allowed. The memory address (binding) hasn't changed.
user = { name: "Charlie" }; // Error. You are trying to change the binding
address.
```

Arrow Functions

- **No `[[Construct]]` Method:** Unlike regular functions, Arrow functions do not have a `[[Construct]]` internal method. This means they **cannot** be used with `new`. This makes them slightly lighter in memory, but more importantly, it enforces their role as pure logic carriers rather than object factories.
- **No `arguments` Object:** They do not have their own `arguments` object. If you access `arguments`, it looks up the scope chain. This is performance-critical in highly nested closures where scope lookup can be expensive.
- **Lexical `this` Binding:** They don't just "bind" `this`; they simply **don't have** a `this`. When you use `this` inside an arrow function, the engine resolves it exactly like it resolves a variable `x`—by looking up the lexical scope.

Spread/Rest Operators

- **Shallow Copy Mechanics:** The spread operator ... performs a shallow copy of **enumerable own properties**. It does *not* copy the prototype chain.
- **The Iterator Protocol:** Spread works on any "Iterable" (objects implementing Symbol.iterator). This is why you can spread a String [...'hello'] but you cannot spread a plain Object into an Array (unless you wrap it).
- **Time Complexity:** Spreading an array or object is an O(n) operation. Doing this inside a loop (e.g., reduce or map) turns a linear algorithm into a quadratic O(n^2) disaster.

```
O(n^2) Complexity
Creating a new array reference AND copying n items in every iteration
const badWay = items.reduce((acc, item) => [...acc, item], []);

O(n) Complexity
Mutation is local and safe here
const goodWay = items.reduce((acc, item) => {
  acc.push(item);
  return acc;
}, []);
```

The JavaScript Runtime

Spread Operator & Performance

Be cautious with { ...state }.

- **Mechanism:** Shallow Copy.
- **Complexity:** O(n) where n is the number of properties.
- **The Trap:** Nested objects are **shared by reference**.

```
const state = {
  config: { theme: 'dark' }, // Reference A
  user: 'Alice'
};
const newState = { ...state }; // Shallow copy
newState.config.theme = 'light';
BUG: state.config.theme is ALSO 'light' now.
You just mutated the original state because 'config' points to Reference A.
```

Array.prototype.map() - The Projection Pattern

A 1-to-1 transformation.

- **Input:** Array of length N.
- **Output:** A *new* Array of length N.
- **The Rule:** You cannot change the size of the array. If you return undefined, the new array will contain undefined at that position.

```
const newArray = array.map((element, index, originalArray) => {
  // Return the new value for this element
  return transform(element);
})
```

Advanced Internals (Memory):

map allocates a brand new array in the Heap. It does not mutate the original array.

- *Performance Note:* If you chain multiple maps (.map().map()), you are allocating and garbage collecting intermediate arrays, which is expensive for large datasets.

Array.prototype.filter() - The Predicate Gate

Subset selection based on a boolean condition (Predicate).

- **Input:** Array of length N
- **Output:** A *new* Array of length 0 to N.
- **Shallow Copy Warning:** filter creates a new array container, but the **items inside are still references**. If you modify an object inside a filtered array, you modify the original object too.

```
const subset = array.filter((element, index, originalArray) => {
  // Return true to keep, false to drop
  return booleanCondition(element);
});
```

Performance Optimization (The Pipeline):

Always filter before you map.

- **Why?** map is expensive (allocating objects, creating JSX).
- **Math:** If you filter 1000 items down to 10, your expensive map only runs 10 times instead of 1000.

Array.prototype.reduce() - The State Deriver

Reducing a list down to a single value (Accumulator).

- **The Power:** It is the super-parent of array methods. You can rewrite map and filter using reduce.
- **The Accumulator:** This is essentially a "state" variable that persists across iterations.

Advanced Pattern: One-Pass Transformation

Instead of iterating twice (filter then map), use reduce to do both in a single pass (O(n)).

```
const finalValue = array.reduce((accumulator, element, index) => {
  // Logic to update the accumulator
  return updatedAccumulator;
}, initialValue);
```

Interview Tip: "Always initialize your accumulator. If you omit the second argument, `reduce` uses the first array item as the accumulator, which crashes your app if the array is empty."

React

What is React?

- **Definition:** A JavaScript **library** for building user interfaces (UI).
- **Origin:** Created by Facebook (Meta) in 2013 to solve the problem of maintaining large, data-heavy applications (like the Facebook News Feed).
- **Key Philosophy:** "Learn Once, Write Anywhere."
  - Web (React.js)
  - Mobile Apps (React Native)
  - VR (React VR)

Library vs. Framework

Framework (e.g., Angular): A strict "full house" solution. It dictates how you write routing, HTTP requests, and state management. Harder to learn, but everything is included.

Library (e.g., React): A "piece of furniture." It only cares about the **View** (the UI). You are free to choose your own router, styling library, etc. It is flexible and lightweight.

React Architecture

Component-Based Architecture

- **Concept:** React breaks a complex UI (like Twitter) into small, reusable blocks called **Components**.
- **Example:**
  - The whole page is the `<App />` component.
  - Inside App, there is a `<Navbar />` and a `<Feed />`.
  - Inside Feed, there are many `<Tweet />` components.
- **Benefit:** If a Tweet breaks, you only fix the Tweet component. You don't break the Navbar.

One-Way Data Flow (Unidirectional)

The Rule: Data always flows **DOWN** like a waterfall.

Parent to Child: A parent passes data to a child via **Props**.

Child to Parent: A child **cannot** push data back up directly. It must communicate via "callbacks" (functions passed down from the parent).

Why? This makes debugging easy. If data is wrong, you know exactly where it came from (upstream).

The Virtual DOM

The Problem: Updating the real browser DOM is slow. It's like rebuilding an entire house just to change a lightbulb.

The Solution: React creates a **Virtual DOM** (a blueprint of the house).

1. **Render:** React updates the Blueprint first (super fast).
2. **Diffing:** React compares the new Blueprint with the old one.
3. **Reconciliation:** React updates the Real House *only* where the change happened (the lightbulb).

What is JSX?

Definition: **J**ava**S**cript **X**ML.

The Concept: It allows us to write HTML-like markup inside a JavaScript file.

The Reality: It is **Syntactic Sugar**.

- Browsers cannot understand JSX.
- A tool called **Babel** transforms it into standard JavaScript (`React.createElement`) that the browser can read.

Takeaway: It looks like HTML, but it has the full power of JavaScript.

Rules

Rule #1 - The Single Parent Rule

- **The Rule:** A component must return **one and only one** parent element.
- **Why?** A JavaScript function cannot return two values at once.

Rule #2 - Closing Tags

- **The Rule:** All tags must be closed. HTML is forgiving; JSX is strict.
- **Self-Closing Tags:** Elements with no children (like images or line breaks) must have a forward slash `/` at the end.
- **Examples:**
  - `<img src="..." />` (Required)
  - `<br />` (Required)
  - `<input type="text" />` (Required)

Rules

Rule #3 - CamelCase Attributes

- **The Conflict:** In HTML, we use class and onclick. In JavaScript, class is a reserved keyword (for creating classes).
- **The Fix:** React uses **camelCase** for attributes.
    - class to className
    - onclick to onClick
    - tabindex to tabIndex
    - for (in labels) to htmlFor

The "Curly Brace" Portal

The Power: This is how you escape from "HTML" back into "JavaScript."

Usage: You can put any valid JavaScript expression inside `{ }`.

Examples:

- **Variables:** `<h1>Hello, {username}</h1>`
- **Math:** `<p>Total: {price * 1.1}</p>`
- **Functions:** `<button>{formatName(user)}</button>`
- **Conditionals:** `{isLoggedIn ? "Logout" : "Login"}`

React Fragments (The Ghost Element)

Problem: Sometimes you don't want to wrap everything in a `<div>` just to satisfy Rule #1 (it messes up CSS grids/flexbox).

Solution: Use a **Fragment**. It groups children without adding an extra node to the DOM.

```
return (
  <>
    <h1>Title</h1>
    <p>Subtitle</p>
  </>
);
```

Props vs. State (The Two Types of Data)

Props (Properties):

- Data passed **into** a component from a parent.
- **Read-Only (Immutable):** A child cannot change its own props.
- *Analogy:* Arguments passed to a function.

State:

- Data managed **inside** the component.
- **Mutable:** The component can change its own state (using `setState` or hooks).
- *Analogy:* Local variables inside a function.

The Component Lifecycle

Every component goes through three phases:

1. **Mounting:** The component is created and inserted into the DOM.
2. **Updating:** The component re-renders because State or Props changed.
3. **Unmounting:** The component is removed from the DOM.

Note: We use `useEffect` to manage actions during these phases.

Functional Components & Props (The "Render Phase")

Pure Functions: A React component is conceptually `UI = f(state)`. It must be idempotent. Calling it multiple times with the same props/state must yield the same JSX result.

Referential Transparency: Since components are functions, props are just arguments. However, React optimizes by checking **Reference Equality** on props.

The "Capture Value" Trait: Functional components capture the *render-time* values of their props and state. This is distinct from Class components, which read from `this.props` (which mutates over time). This difference is the root cause of many "stale UI" bugs in async operations.

Reconciliation (The Diffing Algorithm)

React updates the DOM using a heuristic O(n) algorithm.

1. **Type Check:** If the element type changes (e.g., <div> to <span>), React destroys the old tree and builds a new one.
2. **Keys:** Keys tell React which elements remain stable across renders.
    - **Anti-Pattern:** Using index as a key. If the list order changes, React will reuse DOM nodes incorrectly, leading to input state bugs and performance hits.

State & Reactivity

State (`useState`)

- **Definition:** State is data that changes over time.
- **Variables vs. State:**
  - Changing a variable (`let x = 0; x++`) does **not** trigger a re-render. The UI will not update.
  - Changing state (`setCount(1)`) tells React: "Delete the old UI, paint the new UI."

Anatomy of `useState`

- `const [count, setCount] = useState(0);`
  1. `count`: The current value.
  2. `setCount`: The function to update it.
  3. `0`: The initial value.
- **Rule:** Hooks can only be called at the top level of a component.

Side Effects & Lifecycle

Effects (`useEffect`)

- **The Question:** Where do we put code that connects to the "outside world" (API calls, timers, document title updates)?
- **The Answer:** `useEffect`.
- It synchronizes your component with an external system

The Dependency Array `[ ]`

- Controls *when* the effect runs.
  - `useEffect(fn)` (No array): Runs on **every** render (Dangerous!).
  - `useEffect(fn, [])` (Empty array): Runs **once** on mount (Like `componentDidMount`).
  - `useEffect(fn, [id])`: Runs whenever `id` changes.