React.js Interview Questions and Answers (Expanded In-Depth)

1. Container and Presentational Components

Question: What are Container and Presentational Components?

Answer:

- **Container Components** manage logic, data fetching, and state management.
- **Presentational Components** focus only on UI appearance and receive props.

Deeper In-depth Explanation: Imagine a restaurant: Container components are like chefs (prepare everything), while Presentational components are like servers (just present dishes neatly to customers). This separation improves maintenance and reuse. Containers know "what to do," Presentational components know "how it should look."

Example:

```
function UserContainer() {
  const [user, setUser] = useState(null);
  useEffect(() => { fetchUser().then(setUser); }, []);
  return <UserCard user={user} />;
}

function UserCard({ user }) {
  return <div>{user.name}</div>;
}
```

2. Controlled and Uncontrolled Components

Question: What are Controlled and Uncontrolled Components?

Answer:

- **Controlled**: Form input managed by React state.
- **Uncontrolled**: Form input managed by the DOM (ref).

Deeper In-depth Explanation: Controlled forms let React control the UI fully — validation, error handling, etc. Uncontrolled forms are like "free forms" — faster but harder to validate dynamically. Use Controlled for complex flows, Uncontrolled for simple forms.

Example: Controlled:

```
const [value, setValue] = useState('');
<input value={value} onChange={(e) =>
setValue(e.target.value)} />
```

Uncontrolled:

```
const inputRef = useRef();
<input ref={inputRef} />
```

3. Higher-Order Components (HOC)

Question: What is a Higher-Order Component?

Answer: A HOC is a function that takes a component and returns a new one with enhanced capabilities.

Deeper In-depth Explanation: Think of a HOC like a "gift wrapper". It takes your plain component and adds new features (like loading spinners, permission checks) without touching the core functionality.

Example:

```
function withLoading(Component) {
  return function
WithLoadingComponent({ isLoading, ...props }) {
    if (isLoading) return <h2>Loading...</h2>;
    return <Component {...props} />;
  }
}
```

4. Render Props Pattern

Question: What is the Render Props Pattern?

Answer: A pattern where a component's child is a function that decides what to render.

Deeper In-depth Explanation: Render props make components super flexible. Instead of locking UI logic inside, you allow parent components to "inject" what they want to render based on shared state or behavior.

Example:

```
<MouseTracker render={({ x, y }) => (
  <p>Mouse position: {x}, {y}</p>
)} />
```

5. Compound Components

Question: What are Compound Components?

Answer: Components that internally share state while allowing flexible composition by users.

Deeper In-depth Explanation: Imagine Lego blocks — each block knows how to behave when combined. Compound Components share a hidden context, making it super easy for users to mix and match components together naturally.

Example:

```
<Toggle>
  <Toggle.On>Turned ON</Toggle.On>
  <Toggle.Off>Turned OFF</Toggle.Off>
  <Toggle.Button />
</Toggle>
```

6. Context Provider Pattern

Question: What is the Context Provider Pattern?

Answer: Centralized global state provider to any deeply nested child without prop drilling.

Deeper In-depth Explanation: Context is like a "public announcement" system. Instead of passing props level-by-level, you announce once globally — and any listener (child component) can pick up the data.

Example:

```
<ThemeProvider>
  <App />
</ThemeProvider>

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

7. Custom Hooks

Question: What are Custom Hooks?

Answer: Reusable functions that encapsulate hook logic.

Deeper In-depth Explanation: Custom Hooks let you "package" repetitive logic (like resize tracking, API polling) into reusable mini libraries, keeping components clean and maintainable.

Example:

```
function useWindowSize() {
  const [size, setSize] = useState({ width:
window.innerWidth, height: window.innerHeight });
  useEffect(() => {
    function handleResize() {
      setSize({ width: window.innerWidth, height:
window.innerHeight });
    }
    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize',
handleResize);
  }, []);
  return size;
}
```

8. Error Boundary

Question: What is an Error Boundary?

Answer: Class component that catches JavaScript errors during rendering and shows fallback UI.

Deeper In-depth Explanation: Error Boundaries are your "safety nets". If a part of your UI crashes, the rest keeps running fine. They protect your app from showing the dreaded white screen of death.

Example:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.error("Caught an error:", error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong.</h2>;
    }
    return this.props.children;
  }
}
```

JavaScript Core Topics with In-Depth Explanation (Expanded)

1. Closures

Question: What is a closure in JavaScript?

Answer: Closure is when a function "remembers" its lexical environment.

Deeper In-depth Explanation: Closures let functions access outer variables even after the outer function has finished. They're critical for things like private variables, callback functions, and module patterns.

Example:

```
function outer() {
  let x = 10;
  return function inner() {
    console.log(x);
  };
}
const fn = outer();
fn(); // 10
```

2. Promises and Async/Await

Question: What are Promises and async/await?

Answer: Promises represent eventual success/failure of async code. async/await makes it easier to work with Promises.

Deeper In-depth Explanation: Promises replace messy callback hell. Async/await syntax makes asynchronous code look synchronous, improving readability and error handling.

Example:

```
async function fetchData() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();
  console.log(data);
}
```

3. map, filter, reduce

Question: What are map, filter, reduce?

Answer: Array transformation and reduction methods.

Deeper In-depth Explanation: These methods are core to functional programming in JavaScript. They let you manipulate data immutably without loops, making code shorter and cleaner.

Example:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);
const evens = numbers.filter(n => n % 2 === 0);
const sum = numbers.reduce((acc, n) => acc + n, 0);
```

4. == vs ===

Question: What's the difference between == and ===?

Answer:

- == does type coercion.
- === does strict comparison.

Deeper In-depth Explanation: Always prefer === to avoid bugs. Type coercion often leads to unexpected results.

Example:

```
"5" == 5;  // true
"5" === 5; // false
```

5. Debounce and Throttle

Question: What are debounce and throttle?

Answer: Techniques for limiting function execution frequency.

Deeper In-depth Explanation: Debounce waits for the "silence" after events. Throttle ensures execution at fixed intervals. Crucial for performance in scroll handlers, search boxes.

Example Debounce:

```
function debounce(func, delay) {
  let timeout;
  return function() {
    clearTimeout(timeout);
    timeout = setTimeout(func, delay);
  };
}
```

6. Scope and Hoisting

Question: What are scope and hoisting?

Answer: Scope defines variable visibility. Hoisting moves declarations up.

Deeper In-depth Explanation: Understanding hoisting prevents bugs when accessing variables before declaration. let/const fix many hoisting problems that var had.

Example:

```
console.log(a); // undefined
var a = 10;

console.log(b); // ReferenceError
let b = 20;
```

7. this keyword

Question: What is the 'this' keyword?

Answer: Refers to the object executing the function.

Deeper In-depth Explanation: "this" changes based on how a function is called, not where it's defined. Arrow functions inherit "this" from outer scope.

Example:

```
const obj = {
  name: "JavaScript",
  greet() {
    console.log("Hello, " + this.name);
  }
```

```
};
obj.greet(); // Hello, JavaScript
```

8. Event Loop Basics

Question: How does the JavaScript event loop work?

Answer: Processes synchronous code first, then microtasks, then macrotasks.

Deeper In-depth Explanation: Understanding event loop timing (Promise microtasks before setTimeout macrotasks) is critical for writing performant async code.

Example:

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

Output:

```
Start
End
Promise
Timeout
```

Conclusion

You are now deeply prepared for React and JavaScript interviews with full understanding from fundamentals to advanced patterns. Stay calm, structured, and explain with confidence!