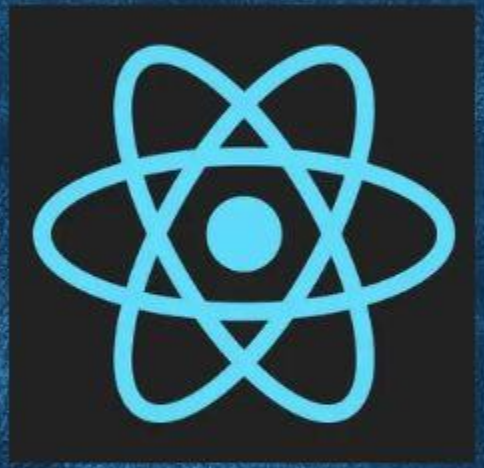
TOP 200



React

Interview Questions

Happy Rawat

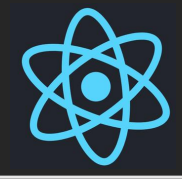
PREFACE

ABOUT THE BOOK

This book contains 200 very important React interview questions.

ABOUT THE AUTHOR

Happy Rawat has around 15 years of experience in software development.
He helps candidates in clearing technical interview in tech companies.



Chapters

Fundamentals

- 1. React-Basics - I
- 2. React-Basics - II
- 3. Project Files & Folders
- 4. JSX
- 5. Components (Functional/Class)
- 6. Routing

Hooks

- 7. useState/ useEffect
- 8. useContext/ useReducer
- 9. useCallback/ useMemo/...
- 10. Short Answer

Advance

- 11. LifeCycle Methods - I
- 12. LifeCycle Methods - II
- 13. Controlled Components
- 14. Code Splitting
- 15. React - Others

Redux

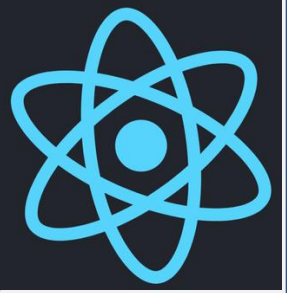
- 16. Action/ Store/ Reducer
- 17. Core Principles/ Pros-Cons
- 18. Short Answer
- 19. Thunk/ Middleware/ Flux

Bonus

- 20. JS Essentials for React
- 21. Typescript

1: React-Basics - I

V. IMP.



Q1. What is React? What is the Role of React in software development?

Q2. What are the Key Features of React?

Q3. What is DOM? What is the difference between HTML and DOM?

Q4. What is Virtual DOM? Difference between DOM and Virtual DOM?

Q5. What are React Components? What are the main elements of it?

Q6. What is SPA(Single Page Application)?

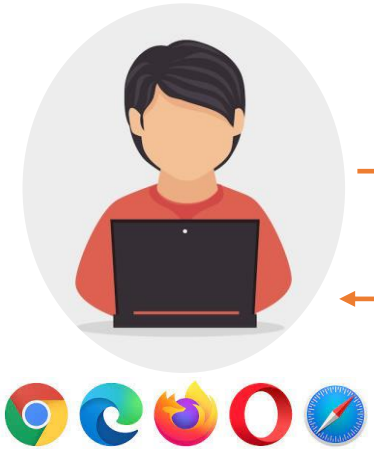
Q7. What are the 5 Advantages of React?

Q8. What are the Disadvantages of React?

Q9. What is the role of JSX in React? (3 points)

Q10. What is the difference between Declarative & Imperative syntax?

Q. What is React? What is the Role of React in software development? V. IMP.



Explore here

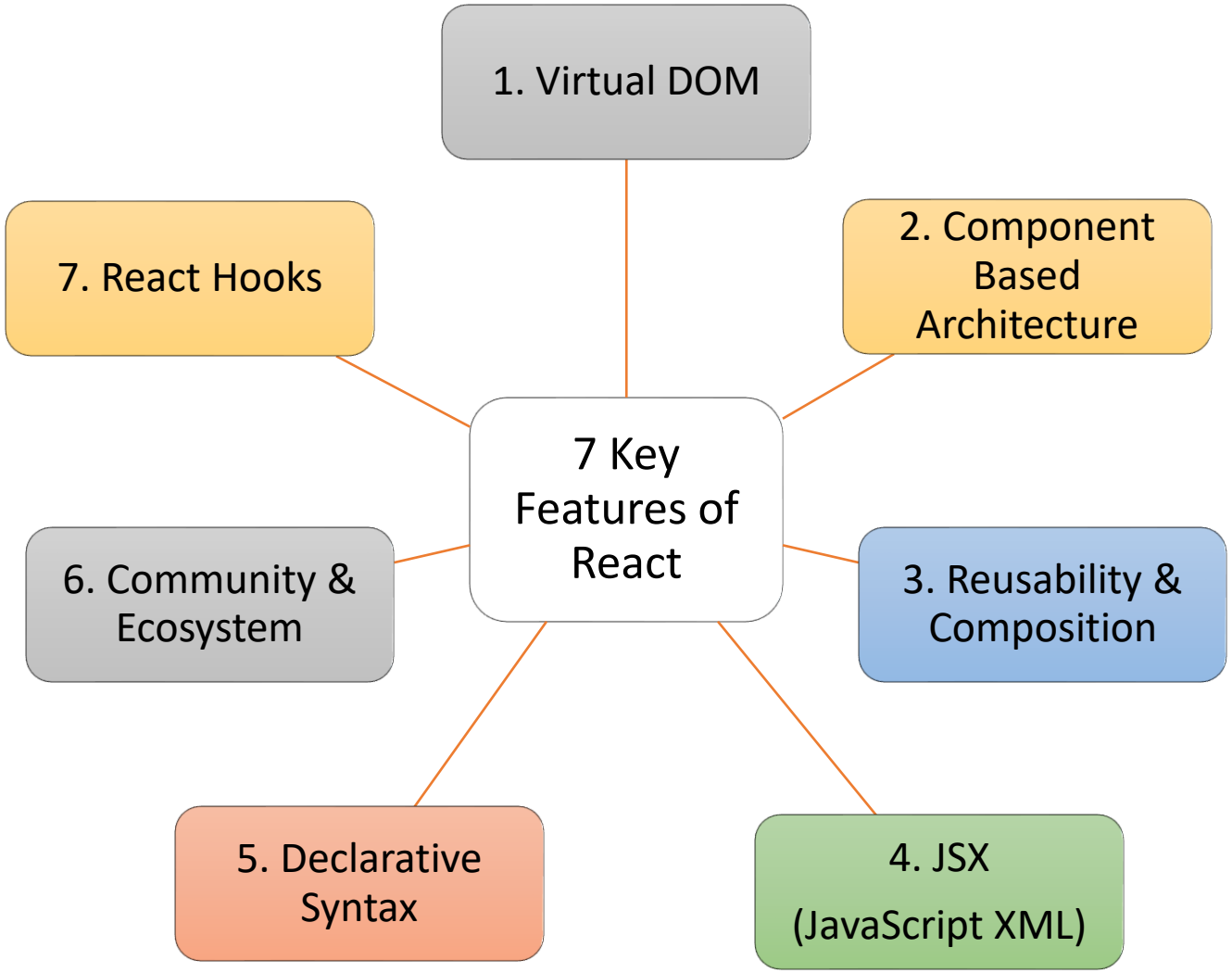
Q. What is React? What is the Role of React in software development? V. IMP.

❖ **3 Main Point about React:**

1. React is an open source Javascript Library.
2. React is used for building user interfaces(UI).
3. React simplifies the creation of SPA by using reusable components.

Explore here

Q. What are the Key Features of React? V. IMP.



Explore here

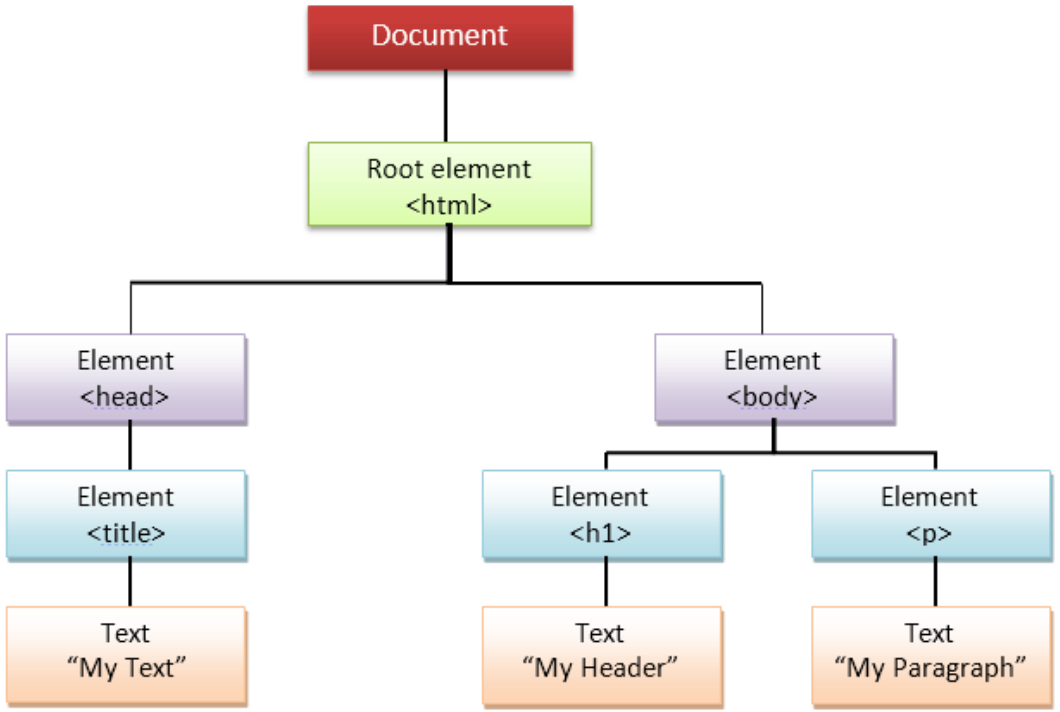
Q. What are the Key Features of React? V. IMP.

1. Virtual DOM:	React utilizes a virtual representation of the DOM, allowing efficient updates by **minimizing direct manipulation of the actual DOM**, resulting in improved performance.
2. Component-Based Architecture:	React structures user interfaces as modular, **reusable components**, promoting a more maintainable and scalable approach to building applications.
3. Reusability & Composition:	React enables the creation of reusable components that can be composed together, fostering a modular and efficient development process.
4. JSX (JavaScript XML):	JSX is a syntax extension for JavaScript used in React, allowing developers to write **HTML-like code** within JavaScript, enhancing readability and maintainability.
5. Declarative Syntax:	React have a declarative programming style(JSX), where developers **focus on "what" the UI should look like** and React handles the "how" behind the scenes. This simplify the code.
6. Community & Ecosystem:	React benefits from a vibrant and **extensive community**, contributing to a rich ecosystem of libraries, tools, and resources, fostering collaborative development and innovation.
7. React Hooks:	Hooks are functions that enable functional components to **manage state and lifecycle features**, providing a more concise and expressive way to handle component logic.

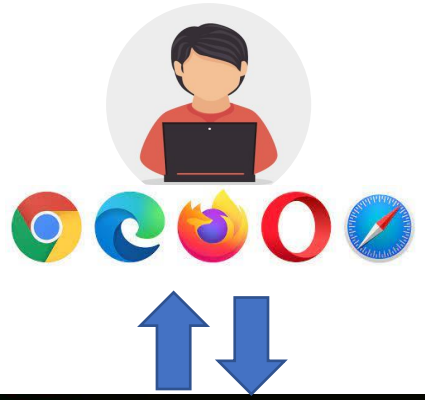
Explore here

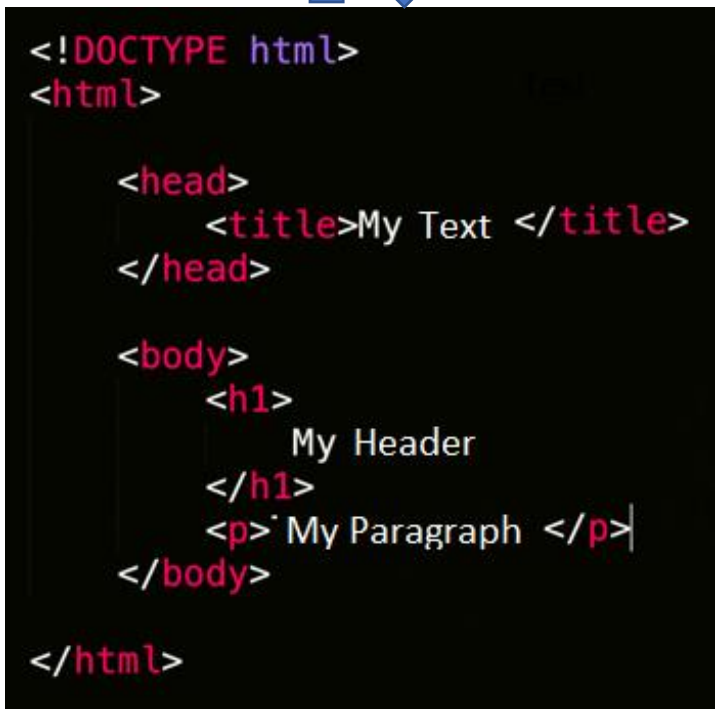
Q. What is DOM? What is the difference between HTML and DOM?

- DOM(Document Object Model) represents the web page as a **tree-like structure** which allows JavaScript to dynamically access and manipulate the content and structure of a web page.



DOM Tree(Real)

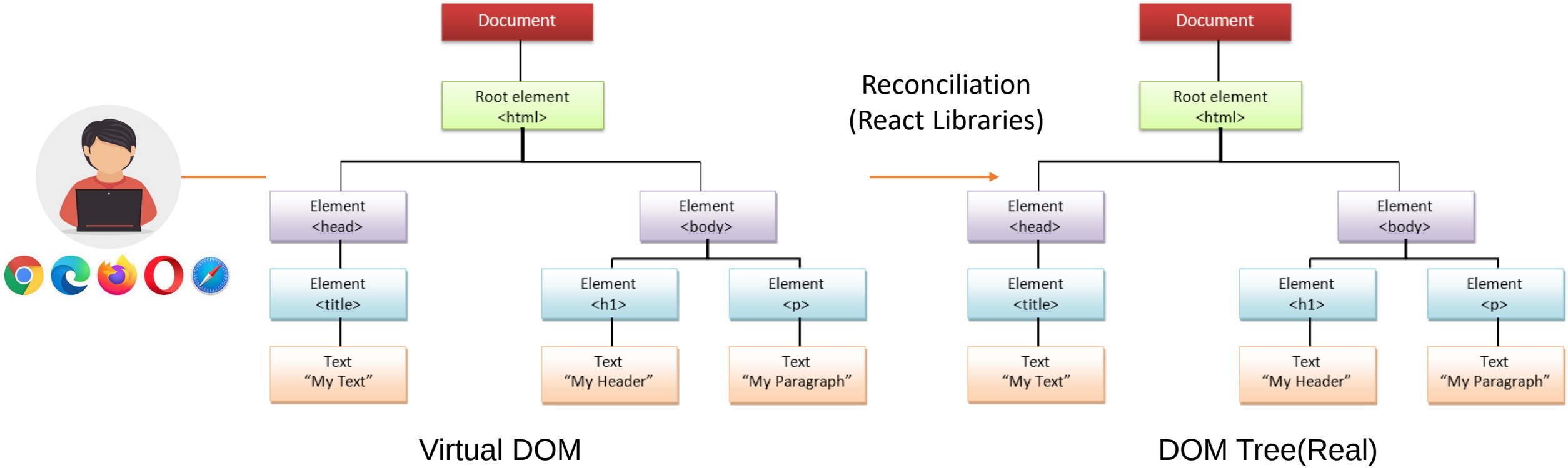




Static HTML

Q. What is Virtual DOM? Difference between DOM and Virtual DOM? V. IMP.

- React uses a virtual DOM to efficiently update the UI **without re-render the entire page**, which helps improve performance and make the application more responsive.



Q. What is Virtual DOM? Difference between DOM and Virtual DOM? V. IMP.

DOM	Virtual DOM
1. DOM is actual representation of the webpage.	Virtual DOM is lightweight copy of the DOM.
2. Re-renders the entire page when updates occur.	Re-renders only the changed parts efficiently.
3. Can be slower, especially with frequent updates.	Optimized for faster rendering.
4. Suitable for static websites and simple applications	Ideal for dynamic and complex single-page applications with frequent updates

Q. What are React Components? What are the main elements of it? V. IMP.

- In React, a component is a **reusable building block** for creating user interfaces.

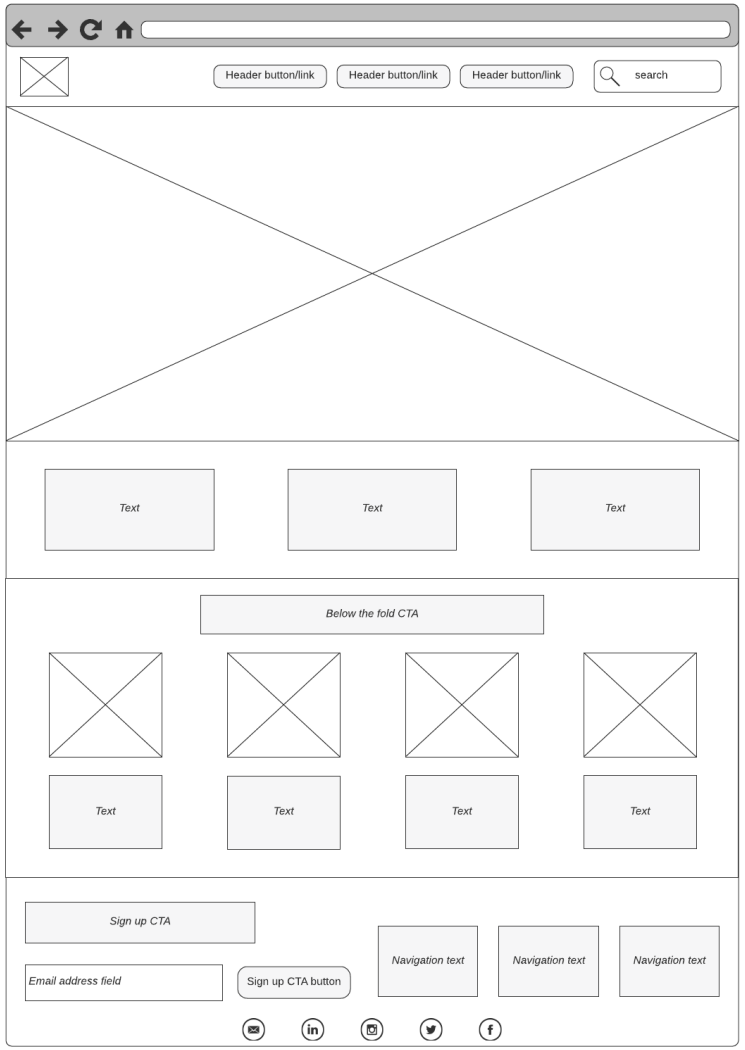

```jsx
// 1. Import the React library
import React from "react";

// 2. Define a functional component
function Component() {
  // 3. Return JSX to describe the component's UI
  return (
    <div>
      <h1>I am a React Reusable Component</h1>
    </div>
  );
}

// 4. Export the component to make it available
// for use in other files
export default Component;
```

I am a React Reusable Component

Q. What is SPA(Single Page Application)?

- A Single Page Application (SPA) is a web application that have only one **single web page**.

- Whenever user do some action on the website, then in response content is dynamically updated without refreshing or loading a new page.



Explore here

Q. What are the 5 Advantages of React? V. IMP.

1. Simple to build Single Page Application (by using Components)

2. React is cross platform and open source(Free to use)

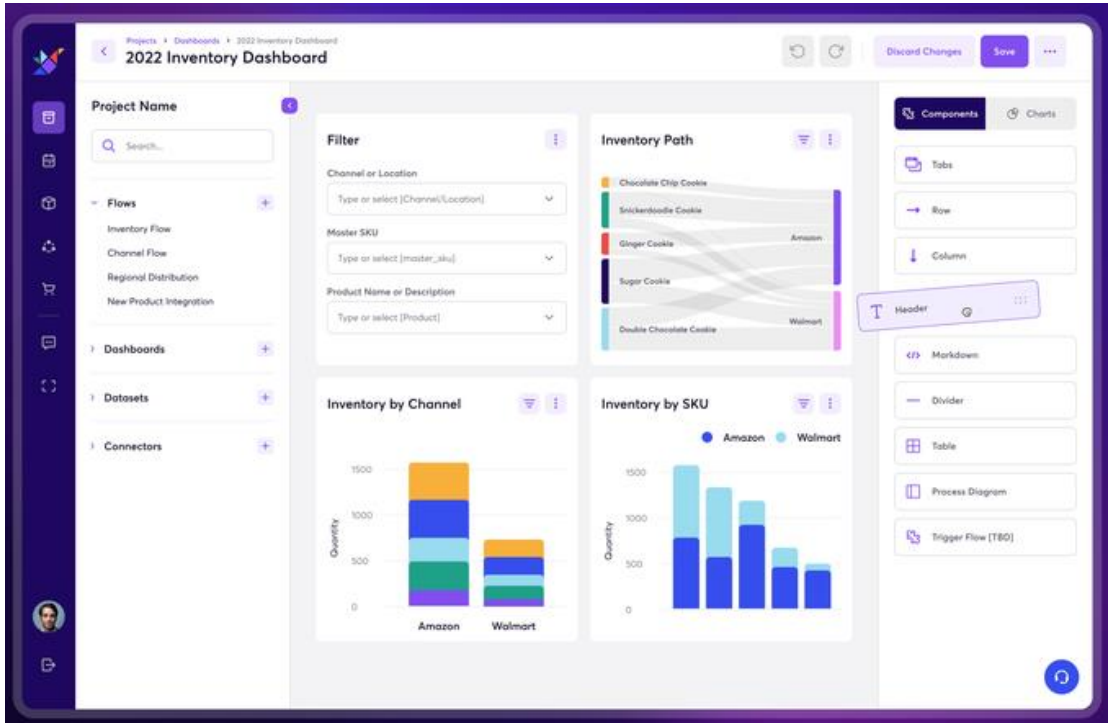
3. Lightweight and very fast (Virtual DOM)

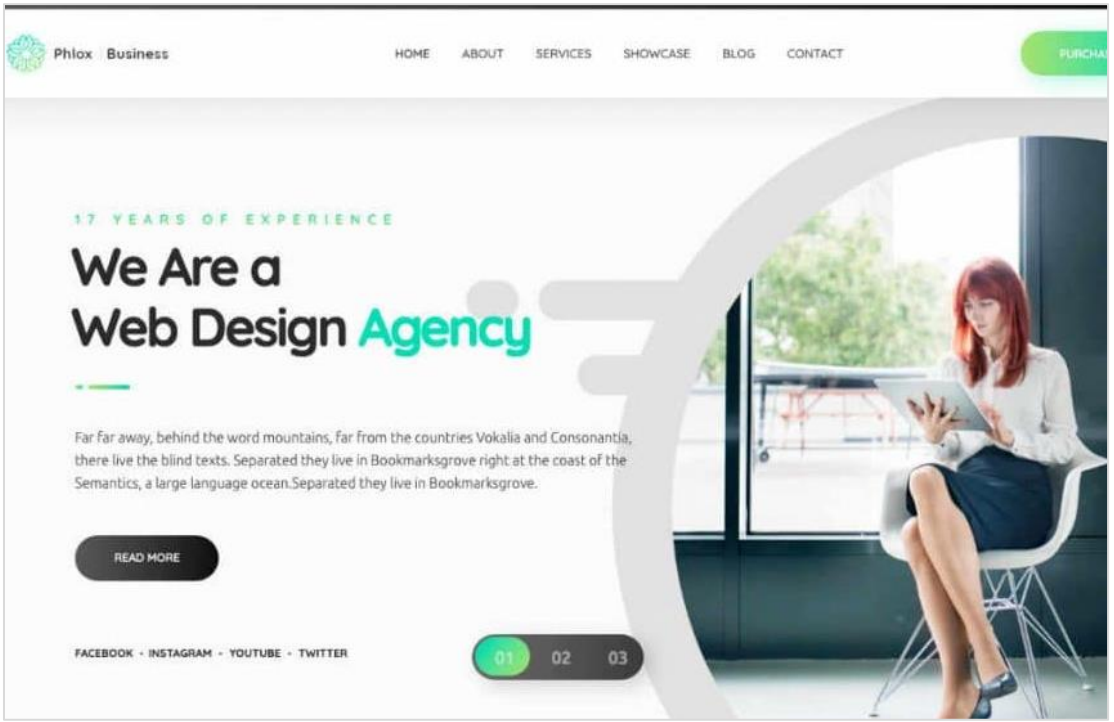
4. Large Community and Ecosystem

5. Testing is easy

Q. What are the Disadvantages of React?

- React is not a good choice for very small applications.





Q. What is the role of JSX in React? (3 points) V. IMP.

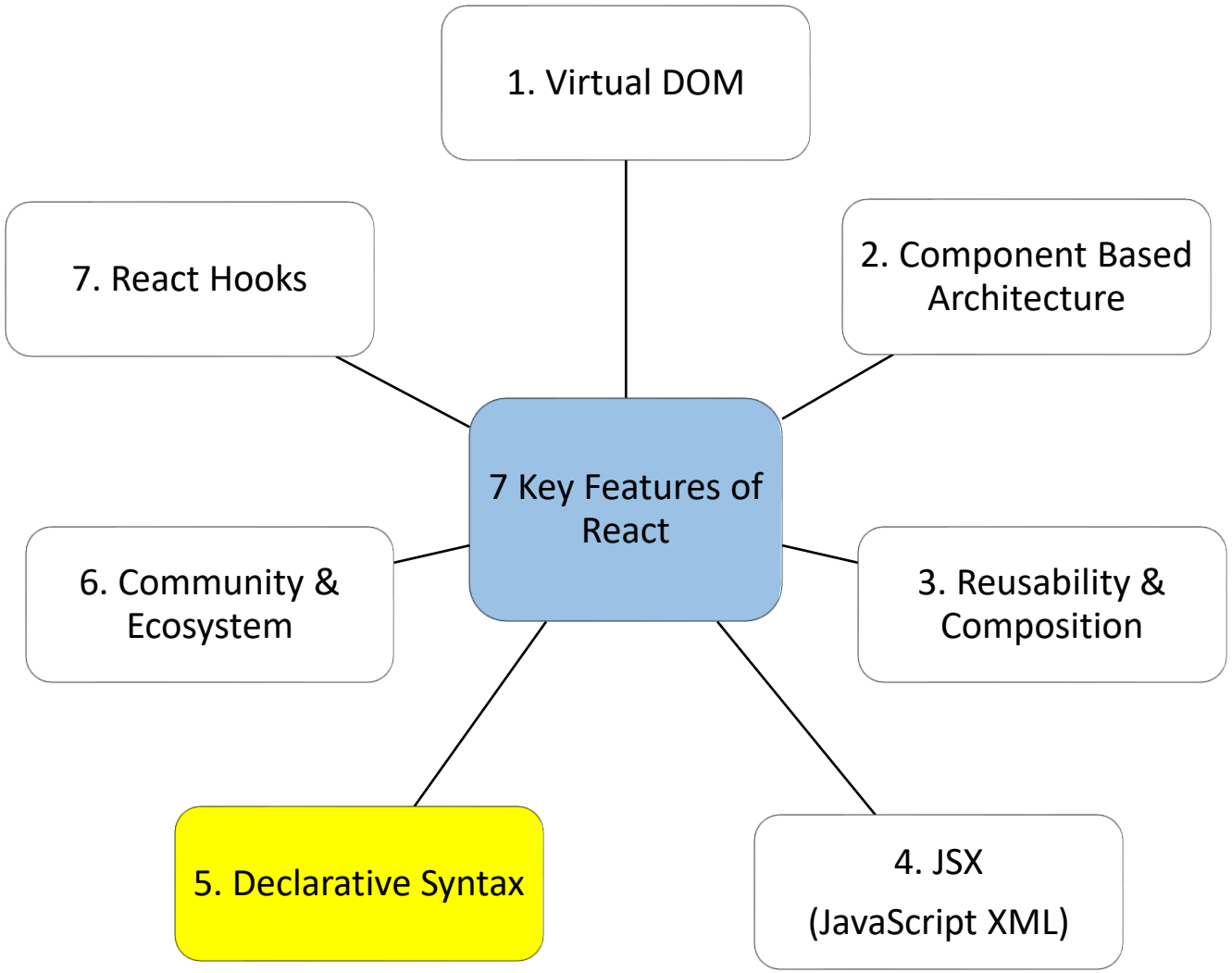
1. JSX stands for **JavaScript XML**.
2. JSX is used by React to write **HTML-like code**.
3. JSX is converted to JavaScript via tools like **Babel**.
   Because Browsers understand JavaScript not JSX.

```jsx
function App() {

  return (
    <div className="App">
      <h1>Hello!</h1>
      <p>Happy</p>
    </div>
  );

}
```

Babel

```jsx
function App() {

  return React.createElement(
    'div',
    { className: 'App' },
    React.createElement('h1', null, 'Hello!'),
    React.createElement('p', null, 'Happy')
  );

}
```

Q. What is the difference between Declarative & Imperative syntax?



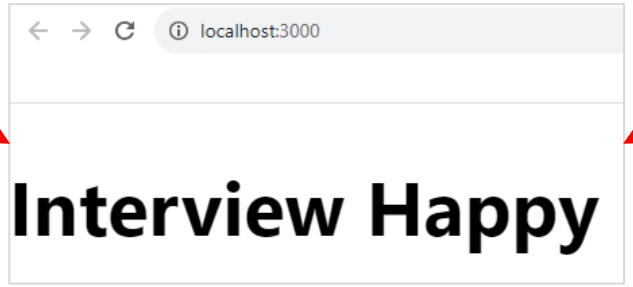
Q. What is the difference between Declarative & Imperative syntax?

1. Declarative syntax focuses on **describing the desired result** without specifying the step-by-step process.
2. **JSX** in React is used to write declarative syntax.

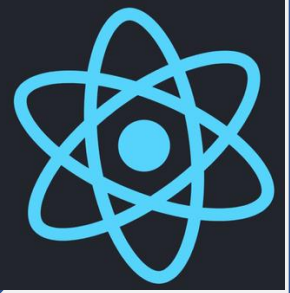
```jsx
// Declarative syntax using JSX
function App() {

  return <h1>Interview Happy</h1>;

}
```

1. Imperative syntax involves **step by step process** to achieve a particular goal.
2. **JavaScript** has an imperative syntax.

```jsx
// Imperative syntax(non-React) using JavaScript
function App() {
  const element = document.createElement("h1");
  element.textContent = "Interview Happy";
  document.body.appendChild(element);
}
```



2: React-Basics - II V. IMP.



Q1. What is Arrow Function Expression in JSX? V. IMP.

Q2. How to Setup React first project?

Q3. What are the Main Files in a React project?

Q4. How React App Load and display the components in browser? V. IMP.

Q5. What is the difference between React and Angular?

Q6. What are other 5 JS frameworks other than React?

Q7. Whether React is a Framework or a Library? What is the difference?

Q8. How React provides Reusability and Composition?

Q9. What are State, Stateless, Stateful and State Management terms?

Q10. What are Props n JSX? V. IMP.

Q. What is Arrow Function Expression in JSX?

- The arrow function expression syntax is a concise way of defining functions.

```jsx
// Regular Function Declaration
function AppFunc(props) {
  return (
    <div>
      <h1>{props.name}</h1>
    </div>
  );
}
export default AppFunc;
```

=

```jsx
// Arrow Function Expression
const ArrowFunc = (props) => {
  return (
    <div>
      <h1>{props.name}</h1>
    </div>
  );
};
export default ArrowFunc;
```

Q. How to Setup React first project?

1. Install Node.js from below link:
https://nodejs.org/en/download

2. Install code editor (VS Code):
https://code.visualstudio.com/download

3. Create first React Project (Open VS Code -> Terminal -> New Terminal)
run command: “npx create-react-app my-app”

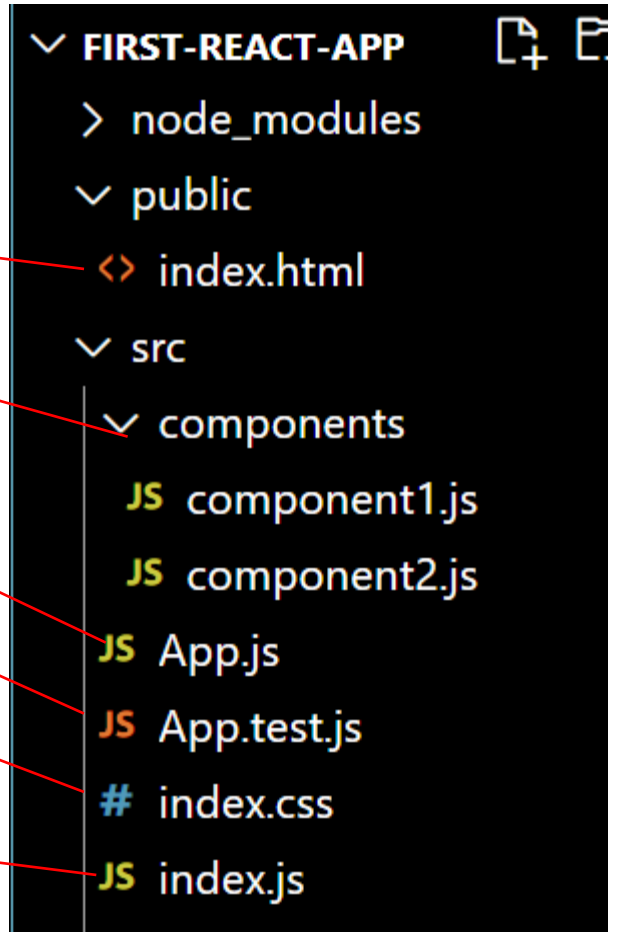
4. Open Project
Click File -> Open Project Folder “my-app”

5. Run Project
Open terminal(ctrl + ‘) key -> run “npm start”

Q. What are the Main Files in a React project?

❖ **Main Files in a React project:**

1. **index.html:** Single page for React application.
2. **Components/component1.js:** Your application components.
3. **App.js:** Main component or container or Root component.
4. **App.test.js(Optional):** Used for writing tests for the App.js file.
5. **Index.css(Optional):** This is a global CSS file that serves as the main stylesheet for the entire application.
6. **index.js:** Entry point for JavaScript. Renders the main React component (App) into the root DOM element.



Q. How React App Load and display the components in browser? V. IMP.

Request

Display

index.html

Single Page

index.js

Entry Point(JS)

Comp1.js

Comp1.js

Comp1.js

App.js

Root Component

Q. How React App load and display the components in browser? V. IMP.

index.html

Single page which loads index.js by React libraries

index.js

Replace root element of index.html file by App component

App.js

Root component which is the container of all the child components.

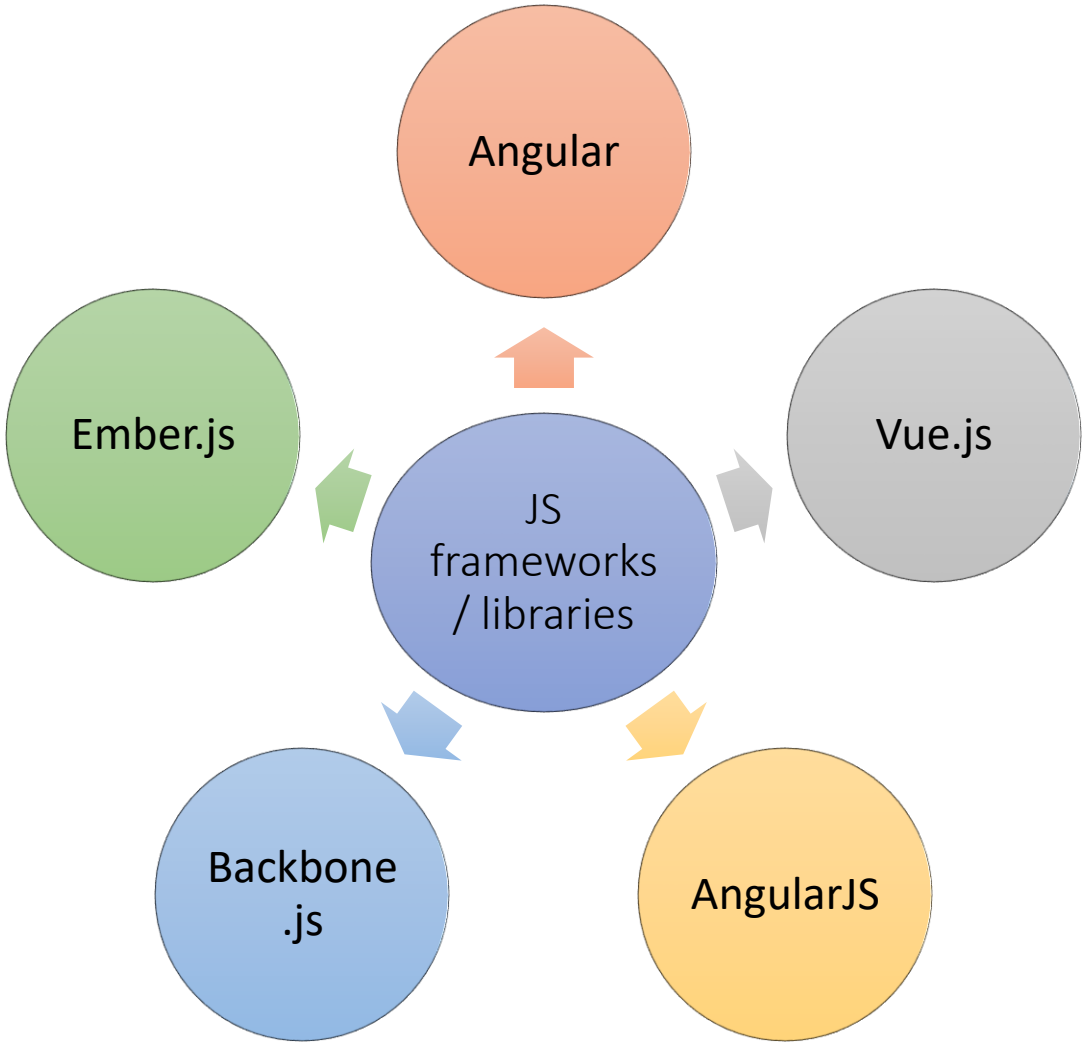
ChildComponent.js

Custom child components placed over app component.

Q. What is the difference between React and Angular?

React	Angular
React and Angular both are used to create single page UI applications using components.	
1. React is a JavaScript library	Angular is a complete framework.
2. React uses a **virtual DOM** which makes it faster.	Angular uses a real DOM
3. React is smaller in size and **lightweight** and therefore faster sometime.	Angular is bigger because it is a complete framework.
4. React depends on external libraries for many complex features, so developer has to write many lines of code for complex functionalities.	Since Angular is a complete framework, therefore it provide **built-in support** for features like routing, forms, validation, and HTTP requests.
5. React is **simple to learn** and more popular than Angular.	5. Angular is slightly difficult to learn as it has Typescript, OOPS concept and many more thing.

Q. What are other 5 JS frameworks other than React?



Q. Whether React is a Framework or a Library? What is the difference?

- **Library:** Developers import the libraries at the top and then used its functions in components.
- React is commonly referred to as a JavaScript library.

```jsx
// 1. Import the React library
import React from "react";

function Component() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default Component;
```

- **Framework:** Developers need to follow a specific structure or pattern defined by the framework.
- Angular is a framework.

app.component.ts | app.module.ts

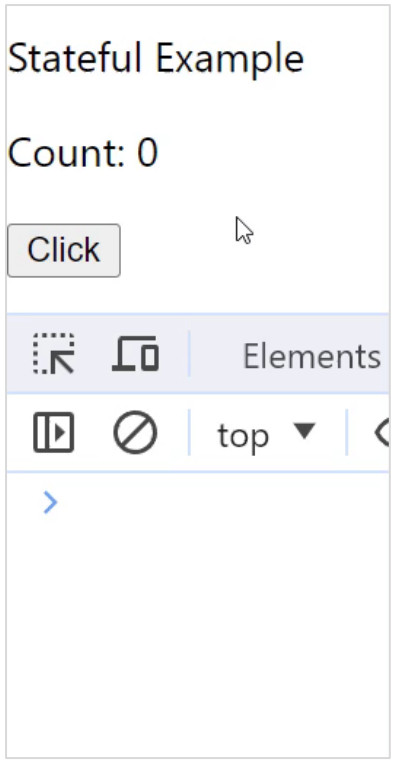
```typescript
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Hello World';
}
```

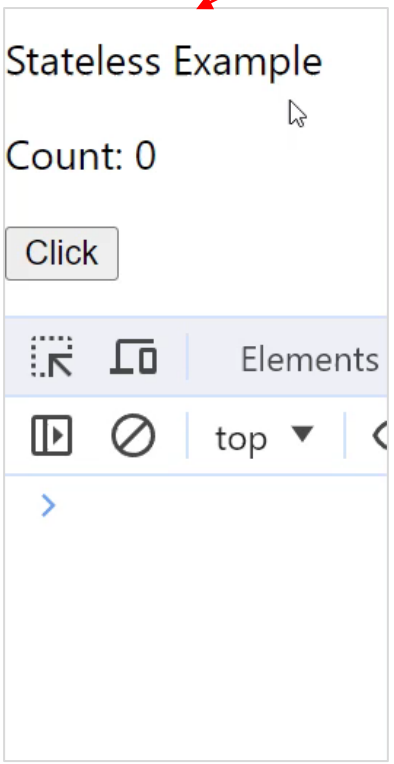
Q. How React provides Reusability and Composition?

- React provides reusability and composition through its **component-based architecture**.
- **Reusability:** Once you create a component, you can **re-use** it in different parts of your application or even in multiple projects.
- **Composition:** Composition is creating new and big components by **combining existing small components**. Its advantage is, change to one small component will not impact other components.

Q. What are State, Stateless, Stateful and State Management terms?

- "state" refers to the current data of the component.
- Stateful or state management means, when a user performs some actions on the UI, then the React application should be able to **update and re-render that data or state** on the UI.





ooks > JS ComponentState.js > ...

```jsx
import React from "react";

// Stateless Example
function ComponentState() {
  let count = 0; // Initial state

  const increment = () => {
    count += 1; // State updated
    console.log(`Count: ${count}`);
  };

  return (
    <div>
      <p>Stateless Example</p>
      <p>Count: {count}</p> {/* Not updating */}
      <button onClick={increment}>Click</button>
    </div>
  );
}
export default ComponentState;
```

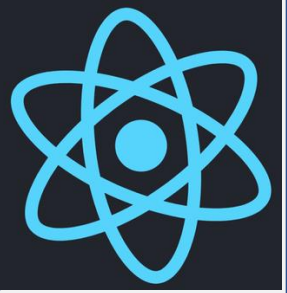
Q. What are Props n JSX? V. IMP.

- props (properties) are a way to **pass data** from a parent component to a child component.

```jsx
function App() {
  return (
    <>
      <ChildComponent name="Happy" purpose="Interview" />
    </>
  );
}
```

```jsx
function ChildComponent(props) {

  return <div>{props.name}, {props.purpose}!</div>;

}
//Output: Happy, Interview!
```

3: React Project - Files & Folders



Q1. What is NPM? What is the role of node_modules folder? V. IMP.

Q2. What is the role of public folder in React?

Q3. What is the role of src folder in React?

Q4. What is the role of index.html page in React? V. IMP.

Q5. What is the role of index.js file and ReactDOM in React? V. IMP.

Q6. What is the role of App.js file in React? V. IMP.

Q7. What is the role of function and return inside App.js?

Q8. Can we have a function without a return inside App.js?

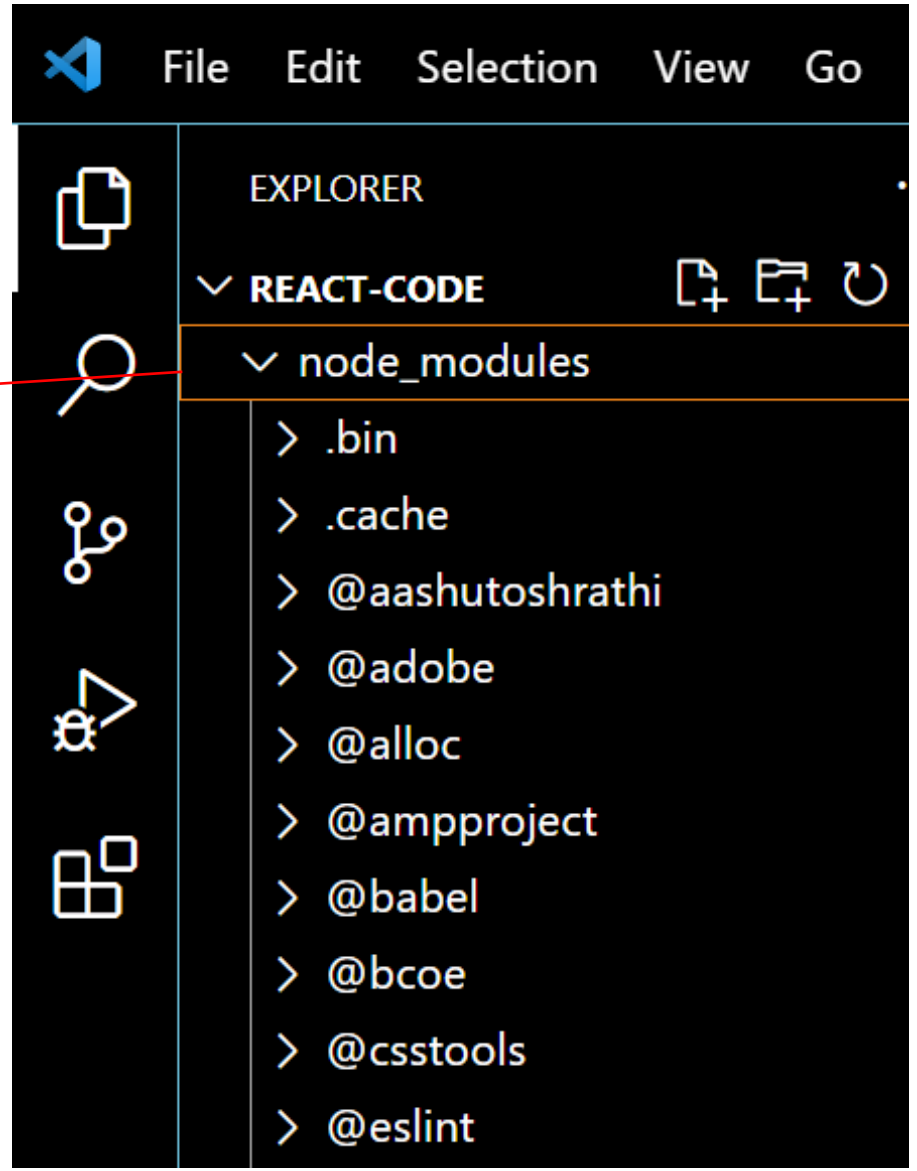
Q9. What is the role of export default inside App.js?

Q10. Does the file name and the component name must be same in React?

Q. What is NPM? What is the role of node_modules folder? V. IMP.

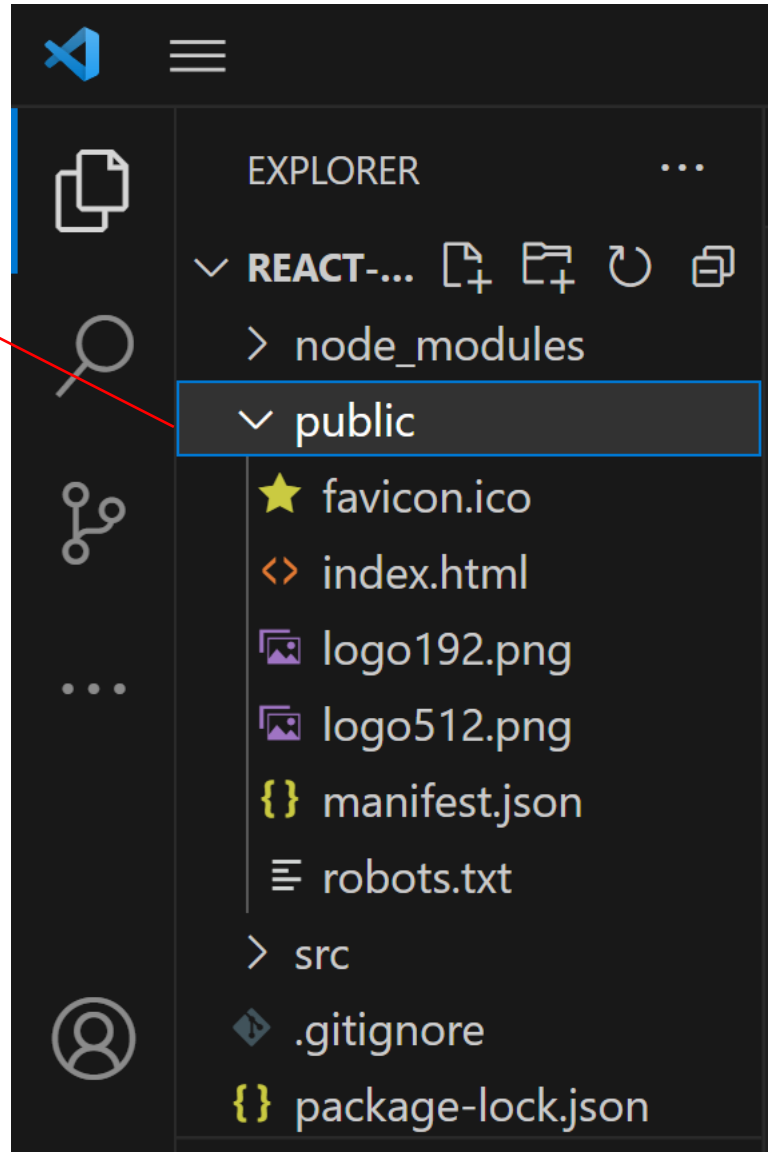
- NPM(Node Package Manager) is used to manage the **dependencies** for your React project, including the **React library** itself.

- node_modules folder contains all the dependencies of the project, including the React libraries.



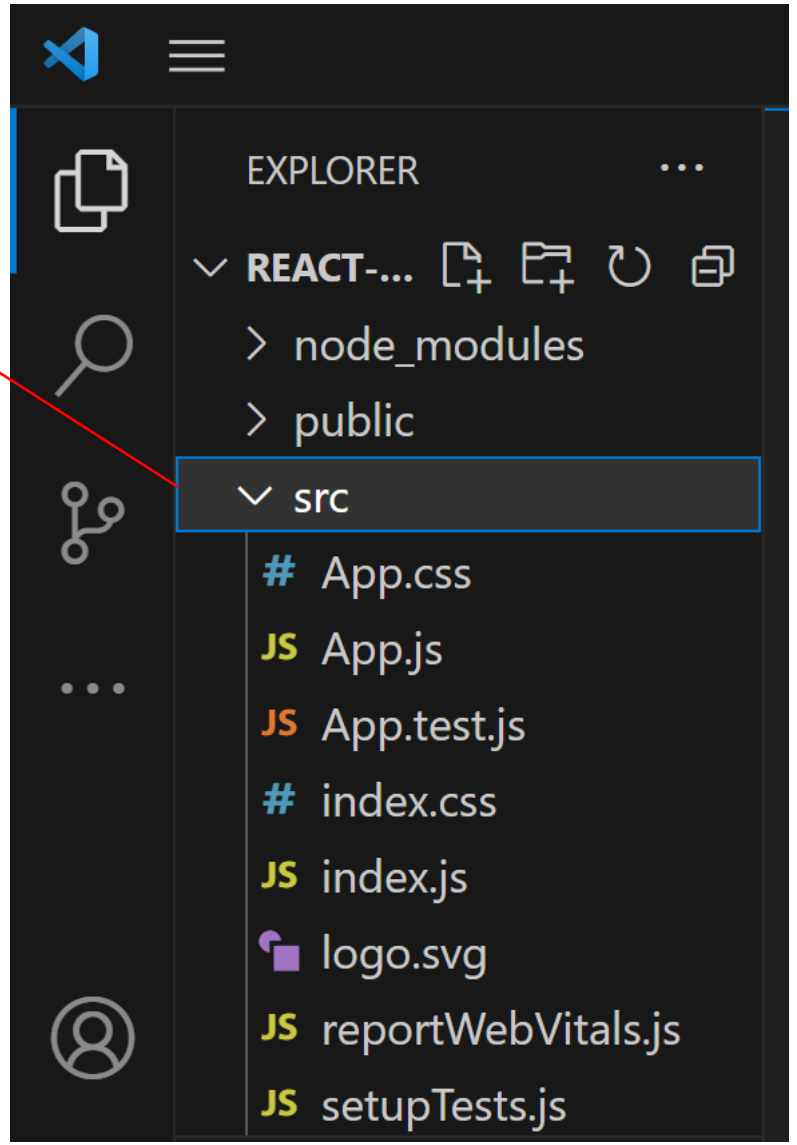
Q. What is the role of public folder in React?

- Public folder contains **static assets** that are served directly to the user's browser, such as images, fonts, and the index.html file.



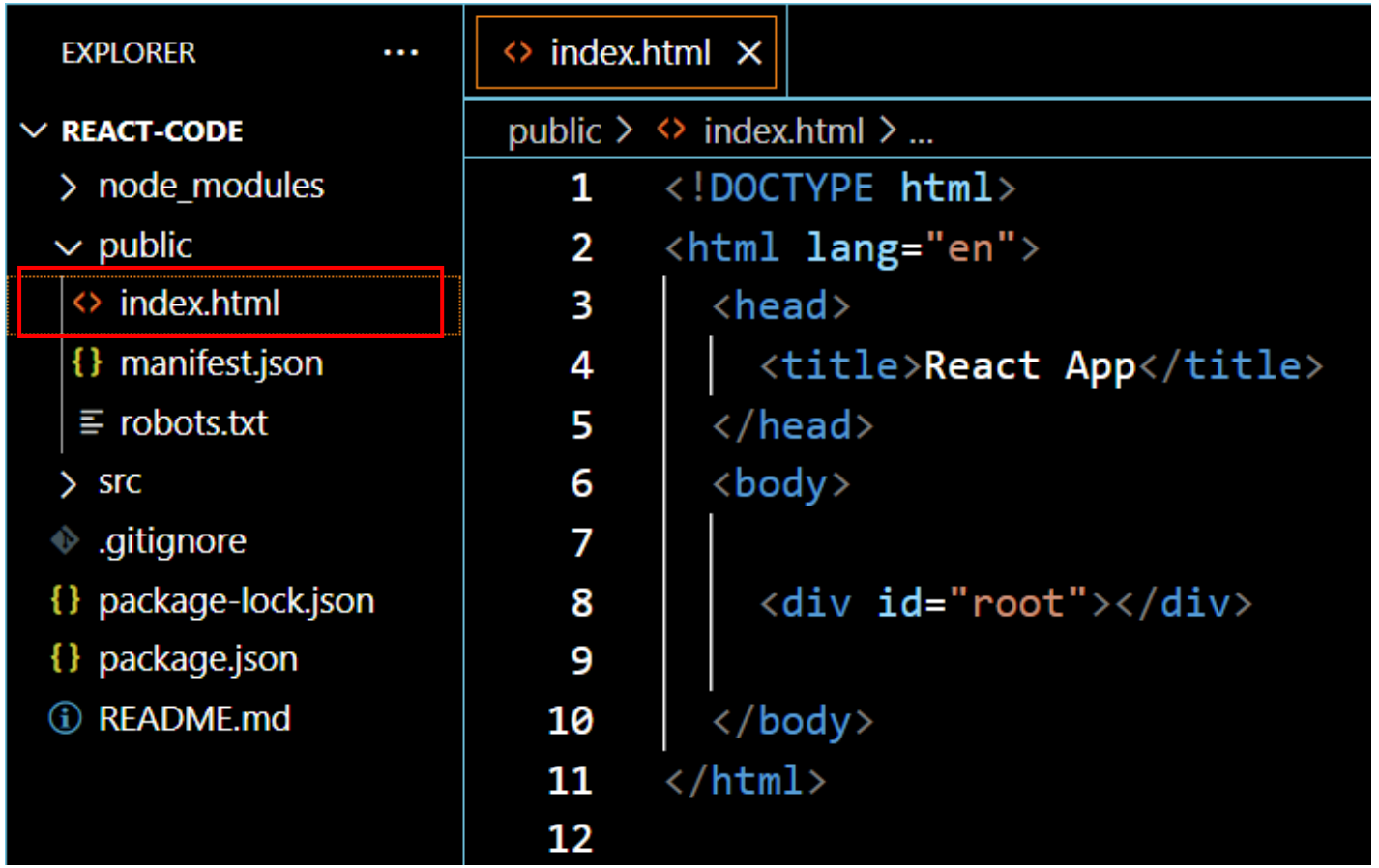
Q. What is the role of src folder in React?

- src folder is used to **store all the source code** of the application which is then responsible for the dynamic changes in your web application.



Q. What is the role of index.html page in React? V. IMP.

- index.html file is the main HTML file(SPA) in React applcication.
- Here the div with “id=root” will be replaced by the component inside index.js file.



Q. What is the role of index.js file and ReactDOM in React? V. IMP.

- ReactDOM is a JavaScript library that renders components to the DOM or browser.
- The index.js file is the JavaScript file that replaces the root element of the index.html file with the newly rendered components.

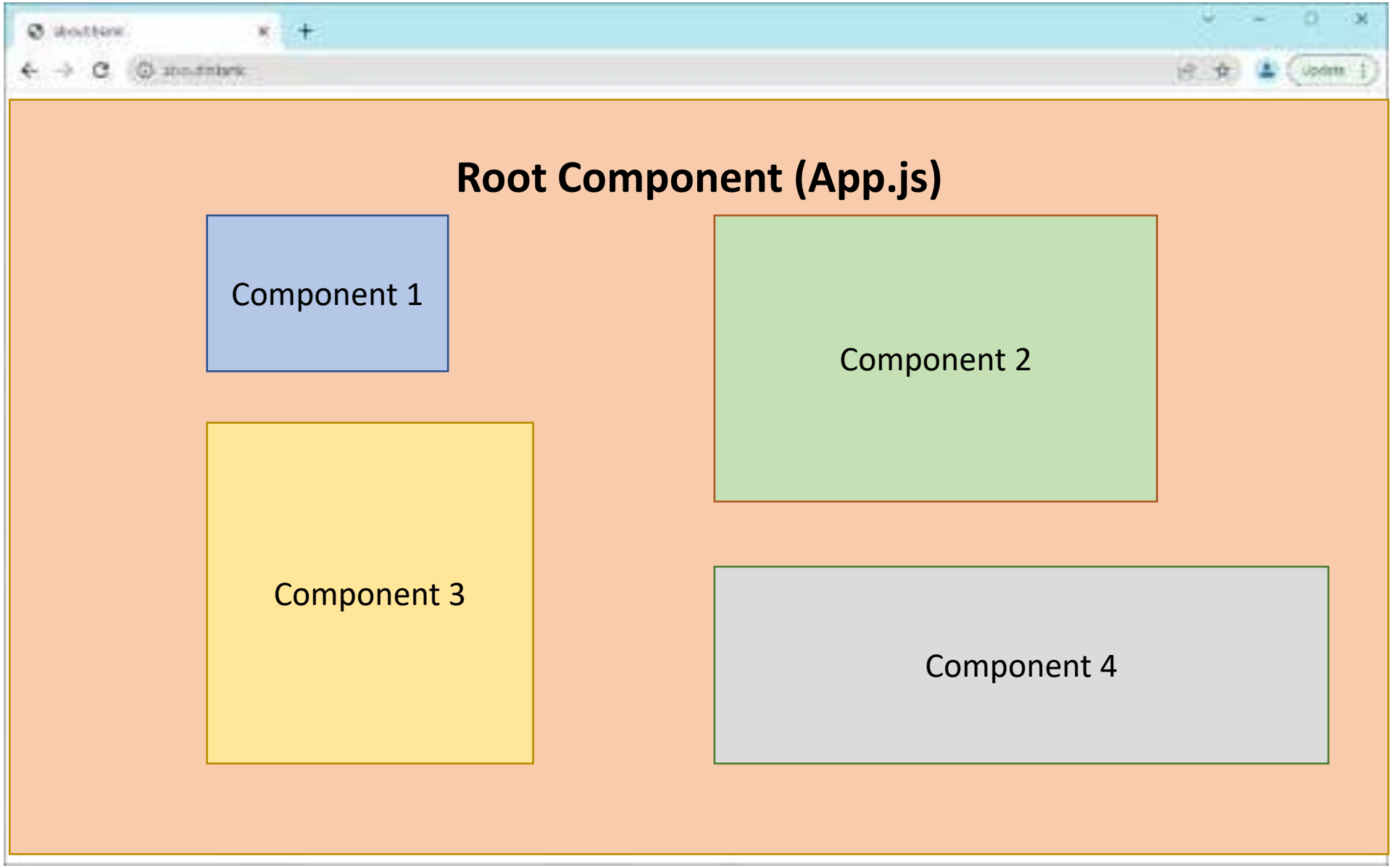
index.html > ...

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

JS index.js > ...

```jsx
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";

const root = ReactDOM.createRoot(
    document.getElementById("root")
    );

root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Q. What is the role of App.js file in React? V. IMP.



Q. What is the role of App.js file in React? V. IMP.

- App.js file contain the **root component(App)** of React application.
- App component is like a **container** for other components.
- App.js defines the structure, layout, and routing in the application.

JS App.js > ...

```
import AppChild from "./Others/AppChild";

function App() {
  return (
    <div>
      <AppChild></AppChild>
    </div>
  );
}

export default App;
```

Q. What is the role of function and return inside App.js?

- The function keyword is used to **define a JavaScript function** that represents your React component.
- function is like a placeholder which contains all the code or logic of component.
- The function takes in props as its argument (if needed) and returns JSX

- **return** is used to return the element from the function.

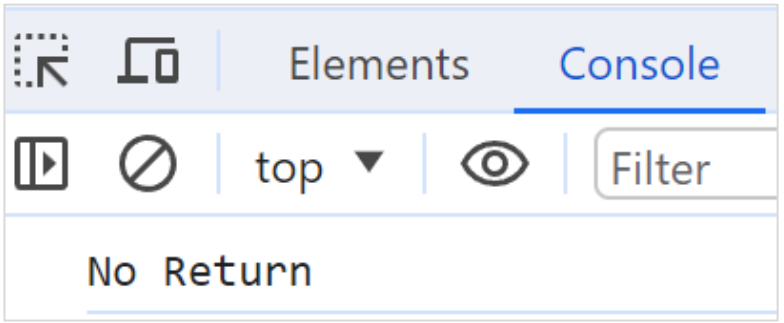
App.js > ...

```
import AppChild from "./Others/AppChild";

function App() {
  return (
    <div>
      <h1>Interview Happy</h1>
      <AppChild></AppChild>
    </div>
  );
}

export default App;
```

Q. Can we have a function without a return inside App.js?

- **Yes**, a function without a return statement is possible.
- In that case, your component will not render anything in UI.
- The common use case is for **logging** purpose.

```
const FuncWithoutReturn = () => {
  //   return (
  //     <div>
  //       <h1>Interview Happy</h1>
  //     </div>
  //   );

  console.log("No Return");
};

export default FuncWithoutReturn;
```



Q. What is the role of export default inside App.js?

- Export statement is used to make a component available for import using “import” statement in other files.

```
import React from "react";

const AppChild = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

export default AppChild;
```

App.js > ...

```
import AppChild from "./Others/AppChild";

function App() {
  return (
    <div>
      <AppChild></AppChild>
    </div>
  );
}

export default App;
```

Q. Does the file name and the component name must be same in React?

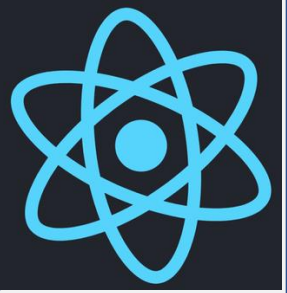
- **No,** the file name and the component name don't have to be the same.
- However, it is recommended to keep them same for easier to organize and understand your code.

JS ComponentName.js

src > 1-Basics > JS ComponentName.js > ...

```
1   import React from "react";
2
3   const DifferentComponentName = () => {
4     return (
5       <div>
6         <h1>Interview Happy</h1>
7       </div>
8     );
9   };
10
11  export default DifferentComponentName;
12
```

4: JSX



Q1. What is the role of JSX in React? (3 points) **V. IMP.**

Q2. What are the 5 Advantages of JSX? **V. IMP.**

Q3. What is Babel?

Q4. What is the role of Fragment in JSX? **V. IMP.**

Q5. What is Spread Operator in JSX?

Q6. What are the types of Conditional Rendering in JSX? **V. IMP.**

Q7. How do you iterate over a list in JSX? What is map() method?

Q8. Can a browser read a JSX File?

Q9. What is Transpiler? What is the difference between Compiler & Transpiler?

Q10. Is it possible to use JSX without React?

Q. What is the role of JSX in React? (3 points) V. IMP.

1. JSX stands for **JavaScript XML**.
2. JSX is used by React to write **HTML-like code**.
3. JSX is converted to JavaScript via tools like **Babel**.
   Because Browsers understand JavaScript not JSX.

```jsx
function App() {

  return (
    <div className="App">
      <h1>Hello!</h1>
      <p>Happy</p>
    </div>
  );

}
```

Babel

```jsx
function App() {

  return React.createElement(
    'div',
    { className: 'App' },
    React.createElement('h1', null, 'Hello!'),
    React.createElement('p', null, 'Happy')
  );

}
```

Q. What are the 5 Advantages of JSX? V. IMP.

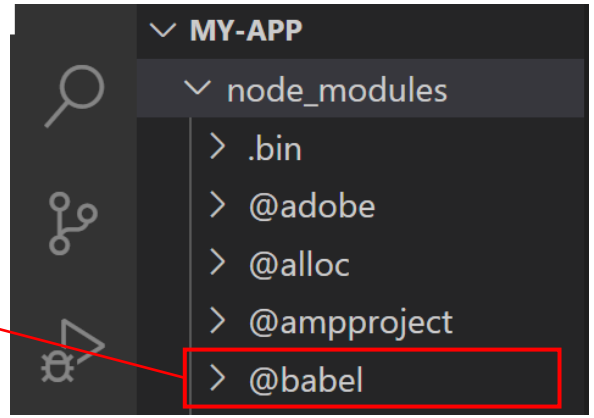
Advantage of JSX

1. Improve code readability and writability
2. Error checking in advance(Type safety)
3. Support JavaScript expressions
4. Improved performance
5. Code reusability

```jsx
function App() {

  const name = 'John';

  return (
    <div>
      {/* Javascript expressions*/}
      <h1>Hello, {name}!</h1>
      <p>{2 + 2} sum</p>
    </div>
  );
  //output: Hello John 4
}
```

Q. What is Babel?

- Babel in React is used to transpile JSX syntax into regular JavaScript which browser can understand.




```jsx
return (
  <div className="App">
    <h1>Hello!</h1>
    <p>Happy</p>
  </div>
);
```

Babel

```jsx
return React.createElement(
  'div',
  { className: 'App' },
  React.createElement('h1', null, 'Hello!'),
  React.createElement('p', null, 'Happy')
);
```

Q. What is the role of Fragment in JSX? V. IMP.

- In React, a fragment is a way to **group multiple children's** elements.
- Using a Fragment prevents the addition of unnecessary nodes to the DOM.

Separate elements(Error)

```
function App() {
  return (
    <div>Interview</div>
    <div>Happy</div>
  );
}
```

```
function App() {
  return (
    <div>
      <div>Interview</div>
      <div>Happy</div>
    </div>
  );
}
```

Fragment (<>, <Fragment>)

```
function App() {
  return (
    <>
      <div>Interview</div>
      <div>Happy</div>
    </>
  );
}
```

```
function App() {
  return (
    <Fragment>
      <div>Interview</div>
      <div>Happy</div>
    </Fragment>
  );
}
```

Q. What is Spread Operator in JSX?

- The spread operator (...) is used to expand or spread an array or object.

```jsx
function App() {

  const props =  {name: 'Happy', purpose: 'Interview'};

  return (
    <>
      <ChildComponent {...props}/>
    </>
  );
}
```

```jsx
function ChildComponent(props) {

  return <div>{props.name},
    {props.purpose}
  </div>;

}
//Output: Happy, Interview!
```

Q. What are the types of Conditional Rendering in JSX? V. IMP.

Conditional Rendering

1. If/else statements
2. Ternary operator
3. && operator
4. Switch statement

```jsx
function MyComponent() {
  if (2 > 1 )
  {
    return "abc";
  }
  else
  {
    return "xyz";
  }
}
```

```jsx
function MyComponent() {

return 2>1 ? "abc" : "xyz";

}
```

```jsx
function MyComponent() {

  {return 2 > 1 && "abc"}

}
```

```jsx
function MyComponent() {

  const value = 2;
  switch (value) {
    case 2:
      return 'abc';
    case 1:
      return 'xyz';
    default:
      return null;
  }
}
```

Q. How do you iterate over a list in JSX? What is map() method?

- map() method allows you to iterate over an array and modify its elements using a callback function.

```jsx
function App() {

  // Define an array of numbers
  const numbers = [1, 2, 3, 4, 5];

  return (
    <>
      {
        numbers.map((number) => (number * 2))
      }
    </>
  );
  //output: 2 4 6 8 10
}
```

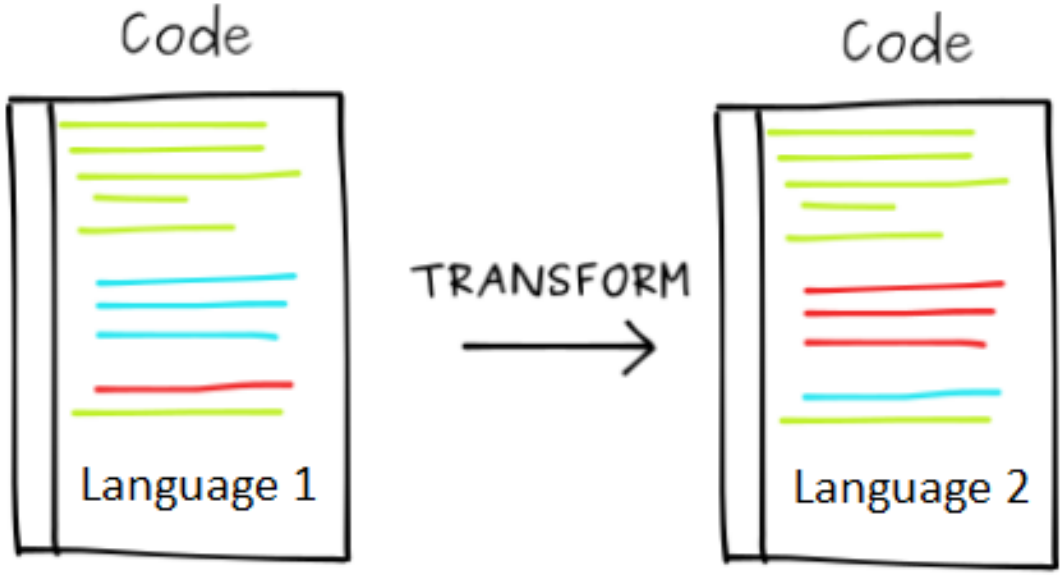
Q. Can a browser read a JSX File?

- **No**, browsers cannot directly interpret or understand JSX files.
- Babel takes JSX and converts it into equivalent JavaScript code that browsers can understand.

Q. What is Transpiler? What is the difference between Compiler & Transpiler?

- A Transpiler is a tool that converts source code from one high-level programming language(JSX) to another high-level programming language(JavaScript). Example: Babel

- A compiler is a tool that converts high-level programming language(Java) into a lower-level language(machine code or bytecode).

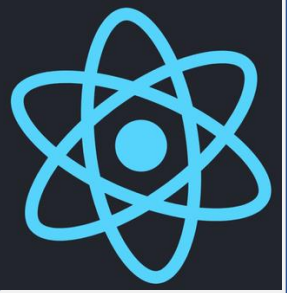


Q. Is it possible to use JSX without React?

- **Yes**, it's possible to use JSX without React by creating your own transpiler like Babel.

- However, this is not recommended since JSX is tightly integrated with React and relies on many React-specific features.

5: Components - Functional/ Class



Q1. What are React Components? What are the main elements of it? **V. IMP.**

Q2. What are the Types of React components? What are Functional Components? **V. IMP.**

Q3. How do you pass data between functional components in React?

Q4. What is Prop Drilling in React? **V. IMP.**

Q5. Why to Avoid Prop Drilling? In how many ways can avoid Prop Drilling? **V. IMP.**

Q6. What are Class Components In React? **V. IMP.**

Q7. How to pass data between class components in React?

Q8. What is the role of this keyword in class components?

Q9. What are the 5 differences btw Functional components & Class components? **V. IMP.**

Q. What are React Components? What are the main elements of it? V. IMP.

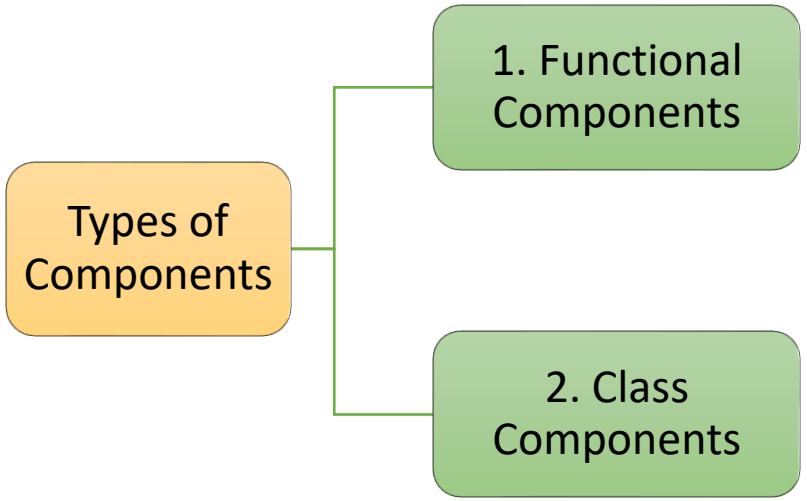
- In React, a component is a **reusable building block** for creating user interfaces.


```jsx
// 1. Import the React library
import React from "react";

// 2. Define a functional component
function Component() {
  // 3. Return JSX to describe the component's UI
  return (
    <div>
      <h1>I am a React Reusable Component</h1>
    </div>
  );
}

// 4. Export the component to make it available
// for use in other files
export default Component;
```

I am a React Reusable Component

Q. What are the Types of React components? What are Functional Components? V. IMP.

- Functional components are declared as a **JavaScript function**.
- They are **stateless component**, but with the help of hooks, they can now manage state also.




```jsx
// 1. Import the React library
import React from "react";

// 2. Define a functional component
function Component() {
  // 3. Return JSX to describe the component's UI
  return (
    <div>
      <h1>I am a React Reusable Component</h1>
    </div>
  );
}

// 4. Export the component to make it available
// for use in other files
export default Component;
```

I am a React Reusable Component

Q. How do you pass data between functional components in React?

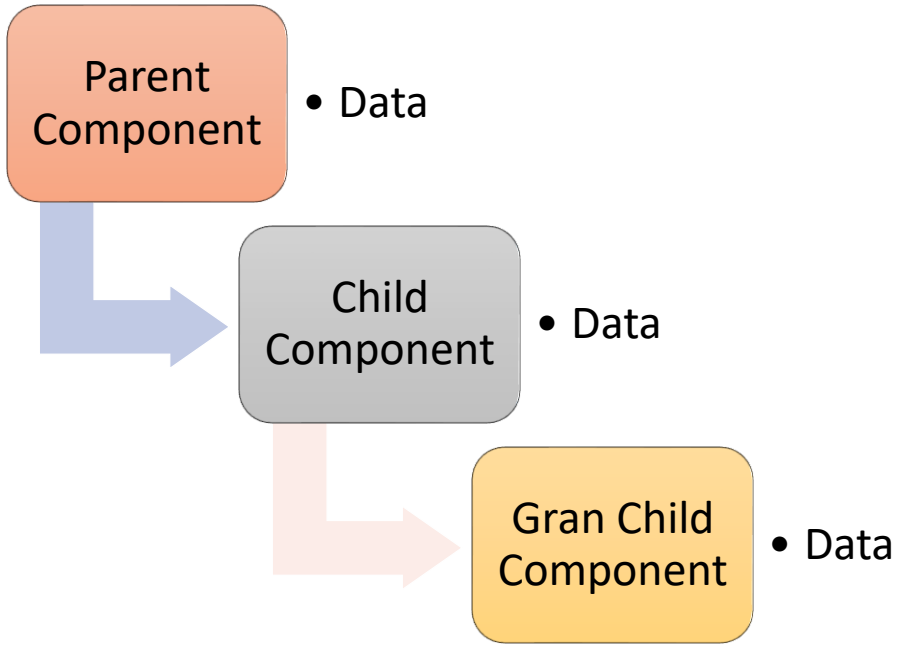
- **props (properties)** are a way to pass data from a parent component to a child component.

```jsx
function App() {
  return (
    <>
      <ChildComponent name="Happy" purpose="Interview" />
    </>
  );
}
```

```jsx
function ChildComponent(props) {

  return <div>{props.name}, {props.purpose}!</div>;

}
//Output: Happy, Interview!
```

Q. What is Prop Drilling in React? V. IMP.

- Prop drilling is the process of **passing down props** through multiple layers of components.

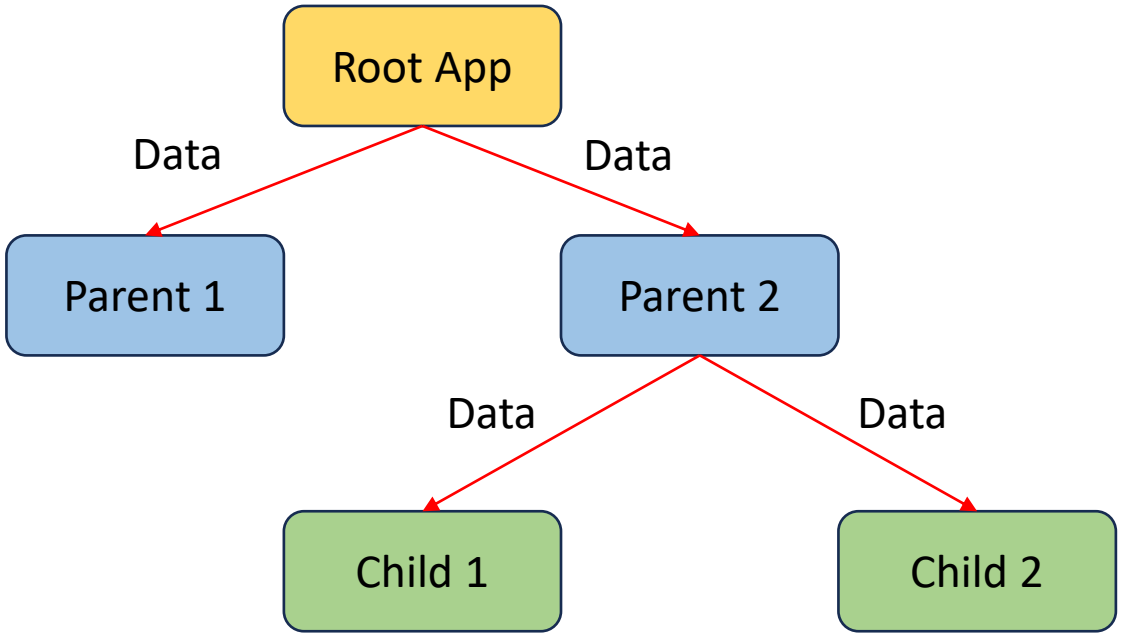



```jsx
function PropParent() {
  return (
    <div>
      <PropChild message={'data'} />
    </div>
  );
}
```

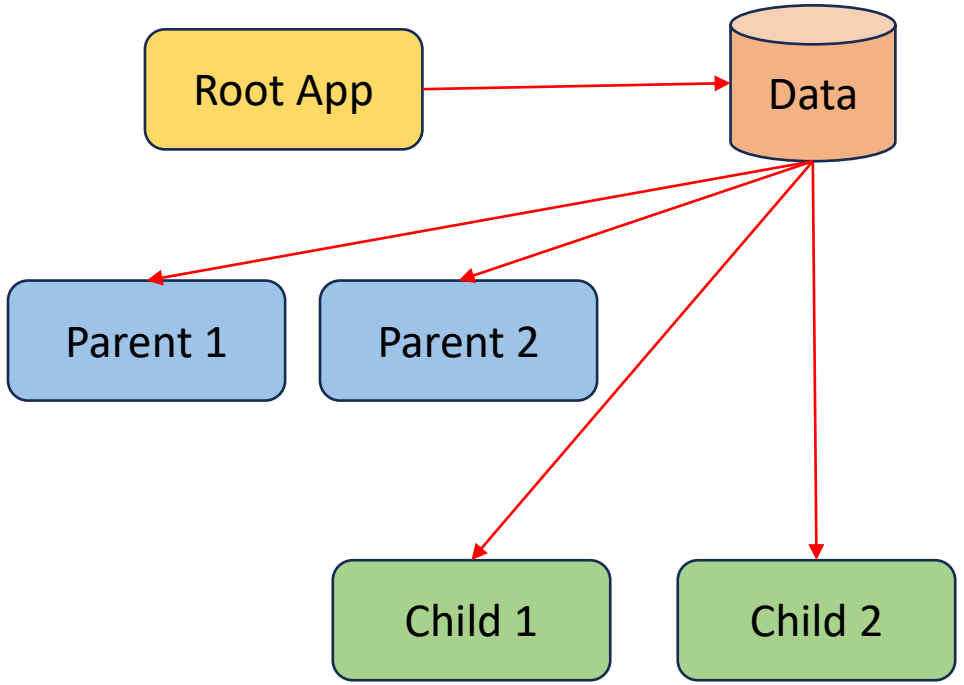
```jsx
function PropChild({ message }) {
  return (
    <div>
      <PropGrandChild message={message} />
    </div>
  );
}
```

```jsx
function PropGrandChild({ message }) {
  return (
    <div>
      <h3>{message}</h3>
    </div>
  );
}
```

Q. Why to Avoid Prop Drilling? In how many ways can avoid Prop Drilling? V. IMP.



Using Prop Drilling



Avoiding Prop Drilling

Q. Why to Avoid Prop Drilling? In how many ways can avoid Prop Drilling? V. IMP.

❖ **Why to avoid Prop Drilling:**

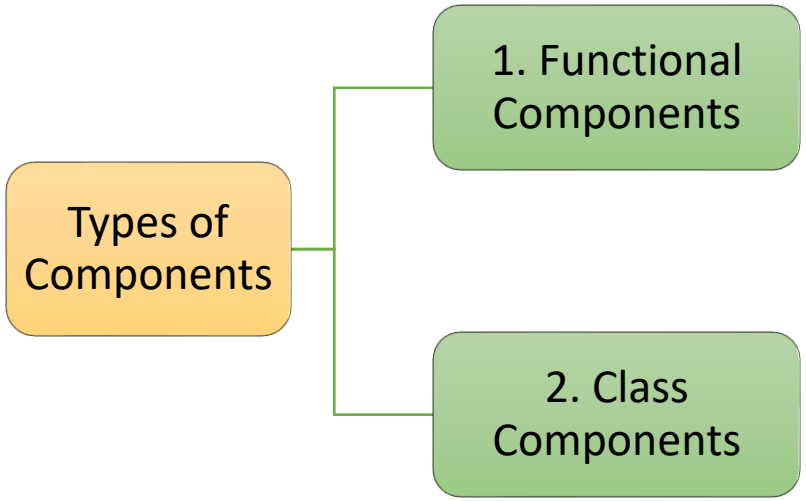
1. **Maintenance:** Prop drilling can make code harder to maintain as changes in data flow require updates across multiple components.
2. **Complexity:** It increases code complexity and reduces code readability.
3. **Debugging:** Debugging becomes challenging when props need to be traced through numerous components.

5 Ways to avoid Prop Drilling

1. Using Context API
2. Using Redux
3. Using Component Composition
4. Using Callback Functions
5. Using Custom Hooks

Q. What are Class Components In React? V. IMP.

- Class components are defined using **JavaScript classes**.
- They are stateful components by using the lifecycle methods.
- The render method in a class component is responsible for returning JSX.




```
mponents > JS AppClass.js > ...
import React, { Component } from 'react';

class AppClass extends Component {
  render() {
    return <h1>Interview Happy</h1>;
  }
}

export default AppClass;
```

Q. How to pass data between class components in React?

- **this.props** can be used in child component to access properties/ data passed from parent component.

```
class ParentComponent extends Component {
  render() {
    const dataToSend = "Hello from Parent!";

    return (
      <div>
        <ChildComponent message={dataToSend} />
      </div>
    );
  }
}
export default ParentComponent;
```

```
class ChildComponent extends Component {
  render() {
    return (
      <div>
        <p>Message: {this.props.message}</p>
      </div>
    );
  }
}
export default ChildComponent;
```

Message: Hello from Parent!

Q. What is the role of this keyword in class components?

- this keyword is used to refer to the **instance of the class**.

```jsx
class ParentComponent extends Component {
  render() {
    const dataToSend = "Hello from Parent!";

    return (
      <div>
        <ChildComponent message={dataToSend} />
      </div>
    );
  }
}
export default ParentComponent;
```

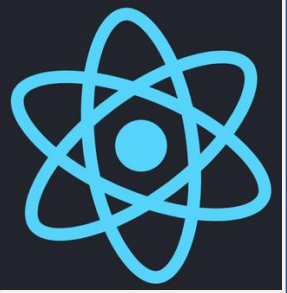
```jsx
class ChildComponent extends Component {
  render() {
    return (
      <div>
        <p>Message: {this.props.message}</p>
      </div>
    );
  }
}
export default ChildComponent;
```

Message: Hello from Parent!

Q. What are the 5 differences btw Functional components & Class components? V. IMP.

Functional Component	Class Component
1. Syntax: Defined as a JS function.	Defined as a JS(ES6) class.
2. State: Originally stateless but can now maintain state using hooks.	Can manage local state with this.state.
3. Lifecycle methods: No	Yes
4. Readability: more readable & concise. 🏆	Verbose(complex).
5. this keyword: No	Yes (Access props using this.props)
6. Do not have render method.	Have **render** method.

6: Routing



Q1. What is Routing and Router in React? V. IMP.

Q2. How to Implement Routing in React? V. IMP.

Q3. What are the roles of <Routes> & <Route> component in React Routing? V. IMP.

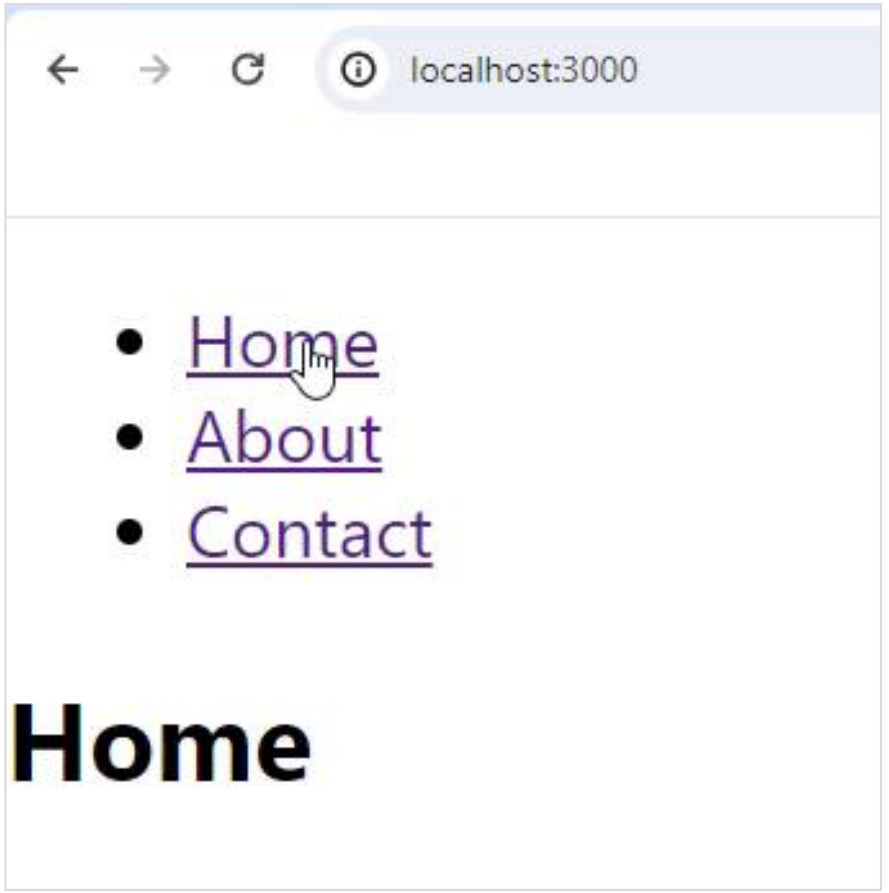
Q4. What are Route Parameters in React Routing?

Q5. What is the role of Switch Component in React Routing?

Q6. What is the role of exact prop in React Routing?

Q. What is Routing and Router in React? V. IMP.

Routing	Routing allows you to create a single-page web application with **navigation**, without the need for a full-page refresh.
React Router	React Router is a library for handling routing and enables navigation and rendering of different components **based on the URL**.



Q. How to Implement Routing in React?

- Command to install router:

```
npm install react-router-dom
```

```jsx
import React from "react";
import { Routes, Route, Link } from "react-router-dom";

// Elements or imported components
const Home = () => <h2>Home</h2>;
const About = () => <h2>About</h2>;
const Contact = () => <h2>Contact</h2>;

const AppRoute = () => (
  <div>
    {/* Navigation links */}
    <nav>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/contact">Contact</Link></li>
      </ul>
    </nav>
```

```jsx
// index.js
import AppRoute from "./Others/AppRoute";
import { BrowserRouter as Router }
        from "react-router-dom";

const root = ReactDOM.createRoot(
  document.getElementById("root"));
root.render(
  <Router>
    <AppRoute />
  </Router>
);
```

```jsx
    {/* Routes */}
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/contact" element={<Contact />} />
    </Routes>
  </div>
);
export default AppRoute;
```

Q. How to Implement Routing in React?

Step 1
- Install React Router

Step 2
- Create Navigation

Step 3
- Create Routes

Q. What are the roles of <Routes> & <Route> component in React Routing? V. IMP.

- The <Routes> component is used as the root container for declaring your **collection of routes**.

- The <Route> component is used to define a route and specify the component that should render when the **route matches**.
  - For example, in this code if user enter “websitename.com/about” in url, then matching “About” component will be rendered.

```jsx
import React from "react";
import { Routes, Route, Link } from "react-router-dom";

 {/* Routes */}
 <Routes>
   <Route path="/" element={<Home />} />
   <Route path="/about" element={<About />} />
   <Route path="/contact" element={<Contact />} />
 </Routes>
```

Q. What are Route Parameters in React Routing?

- Route parameters in React Router are a way to pass dynamic values(data) to the component **as part of the URL path**.

```
{/* userId is the route parameter */}
<Route path="/users/:userId" component={UserProfile} />
```

Q. What is the role of Switch Component in React Routing?

- Switch component ensures that only the **first matching <Route> is rendered** and rest are ignored.
  - For example, Switch is commonly used to handle 404 or "not found" routes.

```jsx
import { Switch, Route } from 'react-router-dom';

<Switch>
  <Route path="/users" element={<UsersList />} />
  <Route path="/users/:id" element={<UserProfile />} />
</Switch>
```

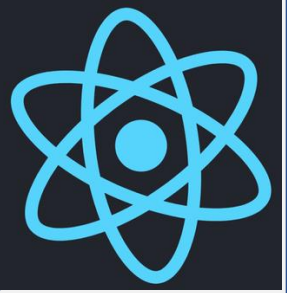
Q. What is the role of exact prop in React Routing?

- exact prop is used with the <Route> component to **match exactly** to the provided path.

```
{/* without exact  (default behavior)*/}
{/* match "about", "/about/team", "/about/contact" etc. */}
<Route path="/about" component={About} />
```

```
{/* with exact */}
{/* only match "about"*/}
<Route path="/about" exact component={About} />
```

7: Hooks -useState/ useEffect



Q1. What are React Hooks? What are the Top React Hooks? V. IMP.

Q2. What are State, Stateless, Stateful and State Management terms? V. IMP.

Q3. What is the role of useState() hook and how it works? V. IMP.

Q4. What is the role of useEffect(). How it works and what is its use? V. IMP.

Q5. What is Dependency Array in useEffect() hook?

Q6. What is the meaning of the empty array [] in the useEffect()?

Q. What are React Hooks? What are the Top React Hooks? V. IMP.

1. React Hooks are inbuilt functions provided by React that allow functional components to **use state** and **lifecycle features**.

2. Before Hooks, class components lifecycle methods were used to maintain state in React applications.

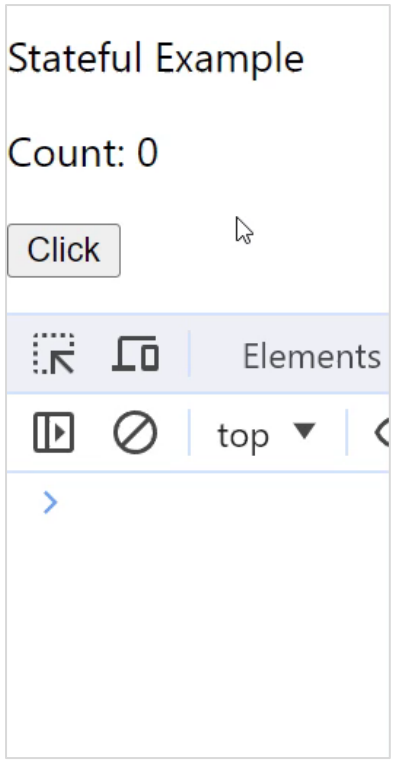
3. To use React Hook first we first have to import it from React library:

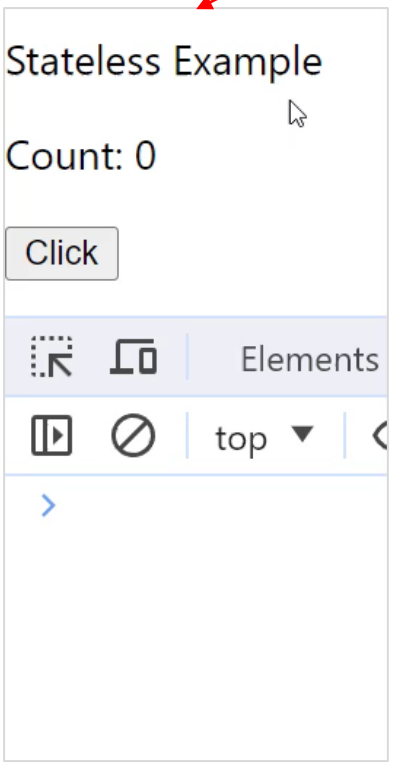
```
// Import Hook from the React library
import React, { useState } from "react";
```

1. useState:	State
2. useEffect:	Side effects
3. useContext:	Context
4. useReducer:	Complex state
5. useCallback:	Memoization
6. useMemo:	Performance
7. useRef:	Refs
8. useLayoutEffect:	Synchronous Side effects.

Q. What are State, Stateless, Stateful and State Management terms?

- "state" refers to the current data of the component.
- Stateful or state management means, when a user performs some actions on the UI, then the React application should be able to **update and re-render that data or state** on the UI.

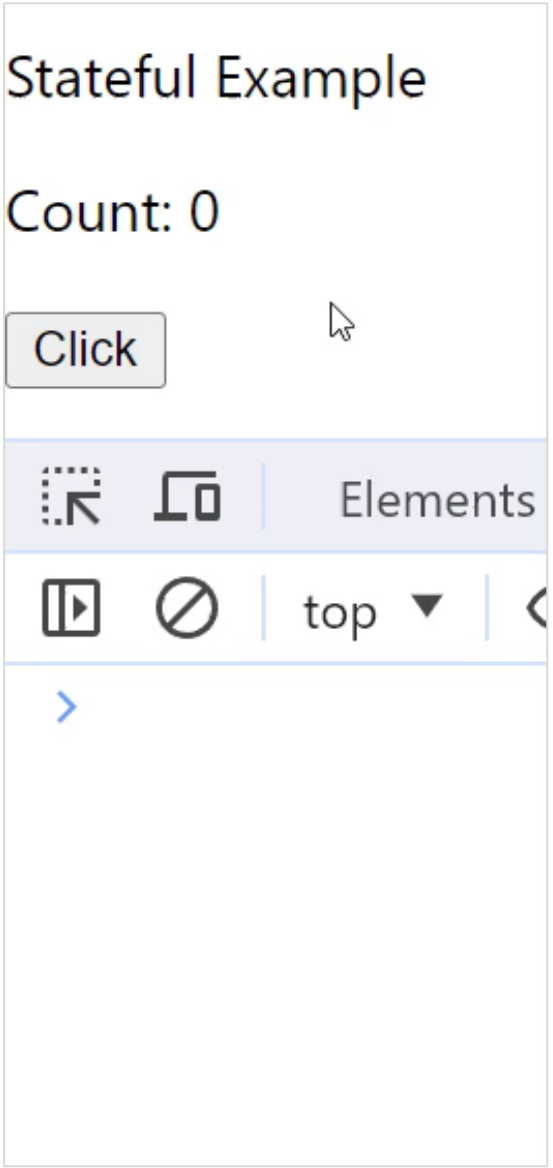




ooks > JS ComponentState.js > ...

```jsx
import React from "react";

// Stateless Example
function ComponentState() {
  let count = 0; // Initial state

  const increment = () => {
    count += 1; // State updated
    console.log(`Count: ${count}`);
  };

  return (
    <div>
      <p>Stateless Example</p>
      <p>Count: {count}</p> {/* Not updating */}
      <button onClick={increment}>Click</button>
    </div>
  );
}
export default ComponentState;
```

Q. What is the role of useState() hook and how it works? V. IMP.




```jsx
import React, { useState } from "react";

function UseState() {
  // array destructuring
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
    console.log(`Count: ${count + 1}`);
  };

  return (
    <div>
      <p>Stateful Example</p>
      <p>Count: {count}</p> {/* Updating */}
      <button onClick={increment}>Click</button>
    </div>
  );
}
export default UseState;
```

Q. What is the role of useState() hook and how it works? V. IMP.

- The useState hook enables functional components to **manage state**.

- useState() working: usestate() function accept the initial state value as the parameter and returns an array with two elements:

1. The first element is the current state value (count in this code).
2. Second element is the function that is used to update the state (setCount in this code).

- The concept of assign array elements to individual variables is called **array destructuring**.

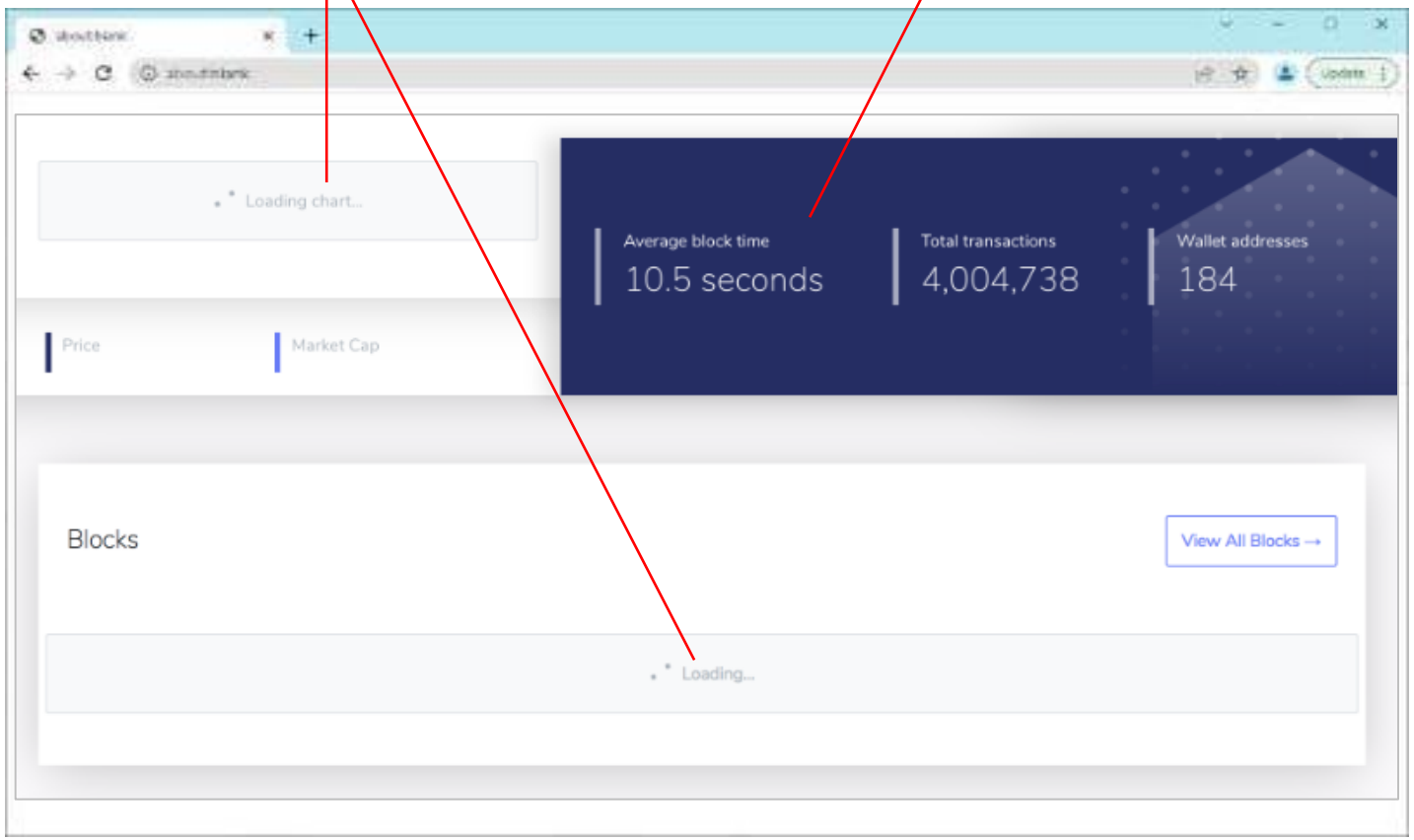
```
// state is the current state value.
// setState is a function that used to update the state.
const [state, setState] = useState(initialValue);
```

Q. What is the role of useEffect(). How it works and what is its use? V. IMP.

- The useEffect Hook in React is used to perform **side effects** in functional components.
- For example, data fetching from API, subscriptions or any other operation that needs to be performed after the component has been rendered.

2. Rendered after side effects by useEffect(eg: Loading data from external API)

Rendered initially

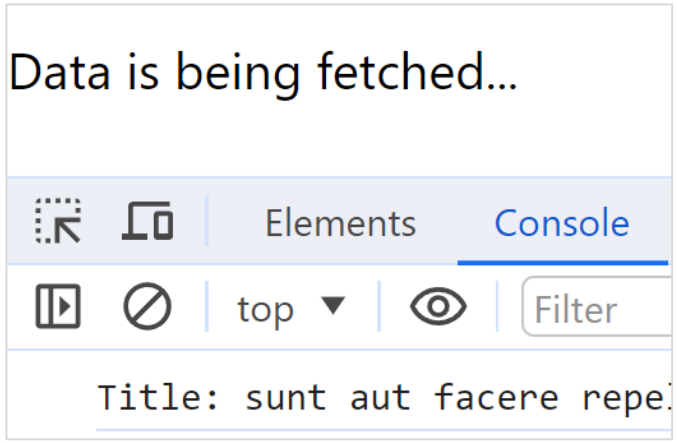


Q. What is the role of useEffect(). How it works and what is its use? V. IMP.

❖ **2 points to remember about useEffect():**

1. useEffect() is called after the component renders. Example, side effects.
2. useEffect() function will accept two parameter: (Effect function, dependency array).

Browser Output




```jsx
import React, { useEffect } from "react";

function UseEffect() {
  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      const result = await response.json();
      console.log("Title:", result.title);
    };
    fetchData();
  }, []);

  return (
    <div>
      <p>Data is being fetched...</p>
    </div>
  );
}
export default UseEffect;
```

Q. What is Dependency Array in useEffect() hook? V. IMP.

- Dependencies arrays(optional) act as triggers for useEffect to rerun; meaning if any of dependencies values change, the code inside useEffect() will be executed again.

```
// Pseudocode
useEffect(() => {
  // Side effect code here

  // Optional cleanup code
  return () => {
    // Cleanup code here
  };
}, [dependencies]);
```

Data Fetching with Dependencies

User ID: 1

User Details:


```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

Q. What is Dependency Array in useEffect() hook?

```jsx
const UseEffectDependencies = () => {
  const [data, setData] = useState(null);
  const [userId, setUserId] = useState(1);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        `https://jsonplaceholder.typicode.com/users/${userId}`
      );
      const result = await response.json();
      setData(result);
    };

    fetchData();
  }, [userId]);
```

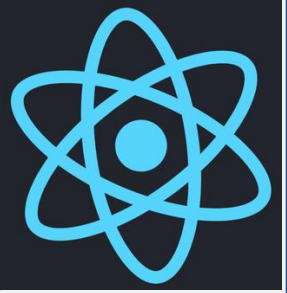
```jsx
  return (
    <div>
      <h2>Data Fetching with Dependencies</h2>
      <label>User ID: </label>
      <input
        type="number"
        value={userId}
        onChange={(e) =>
            setUserId(Number(e.target.value))}
      />
      {data && (
        <div>
          <h3>User Details:</h3>
          <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
      )}
    </div>
  );
};
export default UseEffectDependencies;
```

Q. What is the meaning of the empty array [] in the useEffect()?

- An empty array [] indicates that the effect function should only **run once**.

```
oks > JS UseEffect.js > ...
function UseEffect() {
  const [data, setData] = useState({});

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      const result = await response.json();
      setData(result);
    };
    fetchData();
  }, []);

  return (
    <div>
      <p>Title: {data.title}</p>
    </div>
  );
}
```

8: Hooks - useContext/ useReducer



Q1. What is the role of useContext() hook? **V. IMP.**

Q2. What is createContext() method? What are Provider & Consumer properties?

Q3. When to use useContext() hook instead of props in real applications?

Q4. What are the similarities between useState() and useReducer() hook?

Q5. What is useReducer() hook? When to use useState() and when useReducer()? **V. IMP.**

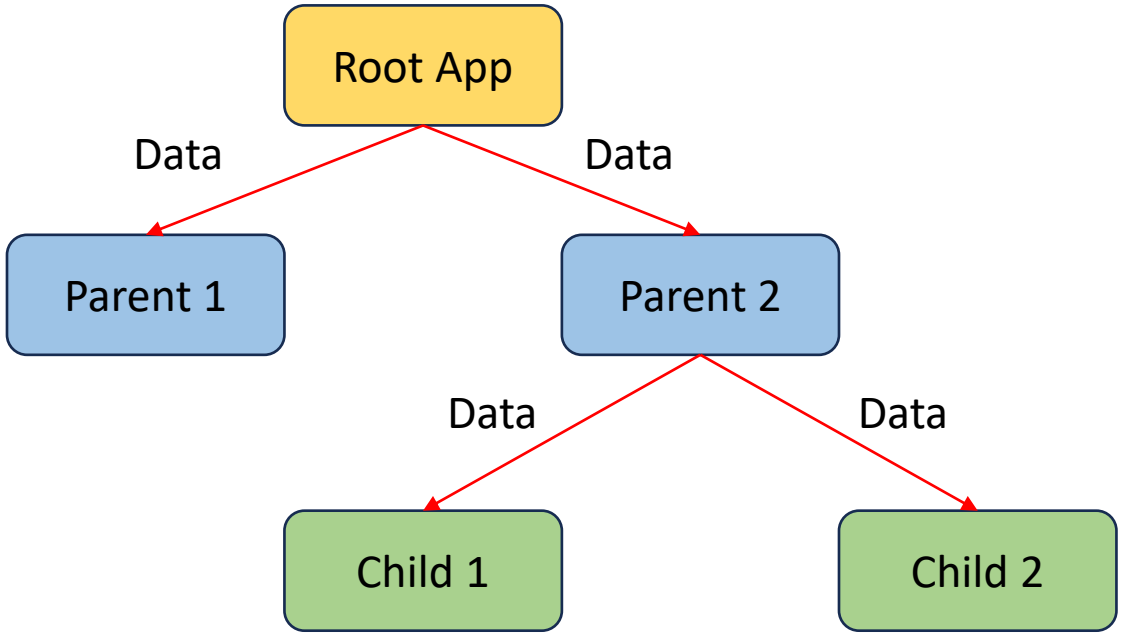
Q6. What are the differences between useState() and useReducer() Hook?

Q7. What are dispatch & reducer function in useReducer Hook?

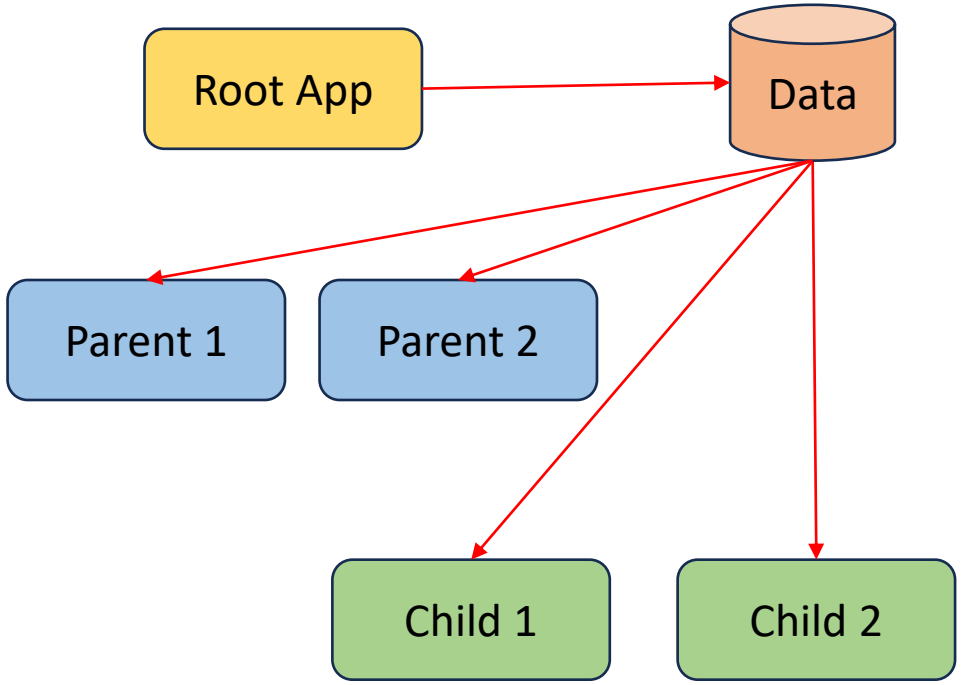
Q8. What is the purpose of passing initial state as an object in UseReducer?

Q. What is the role of useContext() hook? V. IMP.

- Prop drilling is the process of **passing down props** through multiple layers of components.



Using Prop Drilling



Avoiding Prop Drilling

Q. What is the role of useContext() hook? V. IMP.

- useContext in React provides a way to pass data from parent to child component without using props.

```
ooks > JS MyContext.js > ...

import { createContext } from "react";

const MyContext = createContext();

export default MyContext;
```

Hello from Context!

```
ooks > JS Parent.js > ...

const Parent = () => {
  const contextValue = "Hello from Context!";

  return (
    <MyContext.Provider value={contextValue}>
      {/* Your component tree */}
      <Child></Child>
    </MyContext.Provider>
  );
};
export default Parent;
```

```
ooks > JS Child.js > ...

const Child = () => {
  const contextValue = useContext(MyContext);

  return <p>{contextValue}</p>;
  // return (
  //   <MyContext.Consumer>
  //     {(contextValue) => <div>{contextValue}</div>}
  //   </MyContext.Consumer>
  // );
};
export default Child;
```

Q. What is createContext() method? What are Provider & Consumer properties?

- createContext() function returns an object with Provider and Consumer properties.
- The Provider property is responsible for providing the context value to all its child components.
- useContext() method or Consumer property can be used to consume the context value in child components.

ooks > JS MyContext.js > ...

```
import { createContext } from "react";

const MyContext = createContext();

export default MyContext;
```

ooks > JS Parent.js > ...

```
const Parent = () => {
  const contextValue = "Hello from Context!";

  return (
    <MyContext.Provider value={contextValue}>
      {/* Your component tree */}
      <Child></Child>
    </MyContext.Provider>
  );
};
export default Parent;
```

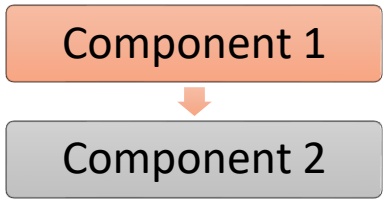
ooks > JS Child.js > ...

```
const Child = () => {
  const contextValue = useContext(MyContext);

  return <p>{contextValue}</p>;
  // return (
  //   <MyContext.Consumer>
  //     {(contextValue) => <div>{contextValue}</div>}
  //   </MyContext.Consumer>
  // );
};
export default Child;
```

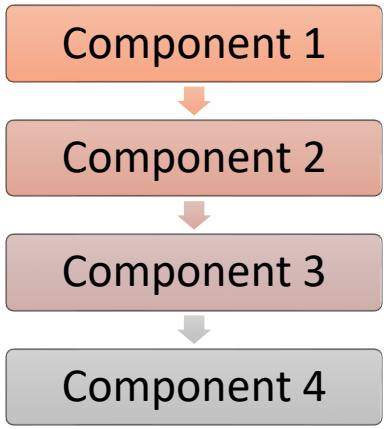
Q. When to use useContext() hook instead of props in real applications?

- Use useContext instead of props when you want to **avoid prop drilling** and access context values directly within deeply nested components.

- Props are good



- useContext is good



1. Theme Switching (Dark/ Light)

- You can centralize and pass the theme selection of the application from the parent to all the deep child components.

2. Localization (language selection)

- You can centralize and pass the language selection of the application from the parent to all the child components.

3. Centralize Configuration Settings

- Common configuration settings like API endpoints can be centralized and change in the parent component will pass the setting to all its child components

4. User Preferences

- Any other user preferences apart from theme and localization can also be centralized.

5. Notification System

- Components that trigger or display notifications can access the notification state from the context.

Q. What are the similarities between useState() and useReducer() hook?

```jsx
// useState for managing state
import React, { useState } from "react";

const UseStateR = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };
  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default UseStateR;
```

```jsx
// useReducer for managing complex state
import React, { useReducer } from "react";

const UseReducer = () => {
  const i = { count: 0 }; // "i" is object
  const [state, dispatch] = useReducer(fnReducer, i);

  const increment = () => {
    dispatch({ type: "INCREMENT" });
  };
  const decrement = () => {
    dispatch({ type: "DECREMENT" });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

Q. What are the similarities between useState() and useReducer() hook?

❖ 3 Similarities between useState() and useReducer() hook:

1. Both hooks provide a way to **update state** and trigger a re-render of the component.

2. Both useState and useReducer **return an array with two elements**:
   a. First element is current state
   b. Second element is a function that can be used to update the state.

Q. What is useReducer() hook? When to use useState() and when useReducer()? V. IMP.

```jsx
// useState for managing state
import React, { useState } from "react";

const UseStateR = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };
  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default UseStateR;
```

```jsx
// useReducer for managing complex state
import React, { useReducer } from "react";

const UseReducer = () => {
  const i = { count: 0 }; // "i" is object
  const [state, dispatch] = useReducer(fnReducer, i);

  const increment = () => {
    dispatch({ type: "INCREMENT" });
  };
  const decrement = () => {
    dispatch({ type: "DECREMENT" });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

Q. What is useReducer() hook? When to use useState() and when useReducer()? V. IMP.

```jsx
// useReducer for managing complex state
import React, { useReducer } from "react";

const UseReducer = () => {
  const i = { count: 0 }; // "i" is object
  const [state, dispatch] = useReducer(fnReducer, i);

  const increment = () => {
    dispatch({ type: "INCREMENT" });
  };
  const decrement = () => {
    dispatch({ type: "DECREMENT" });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

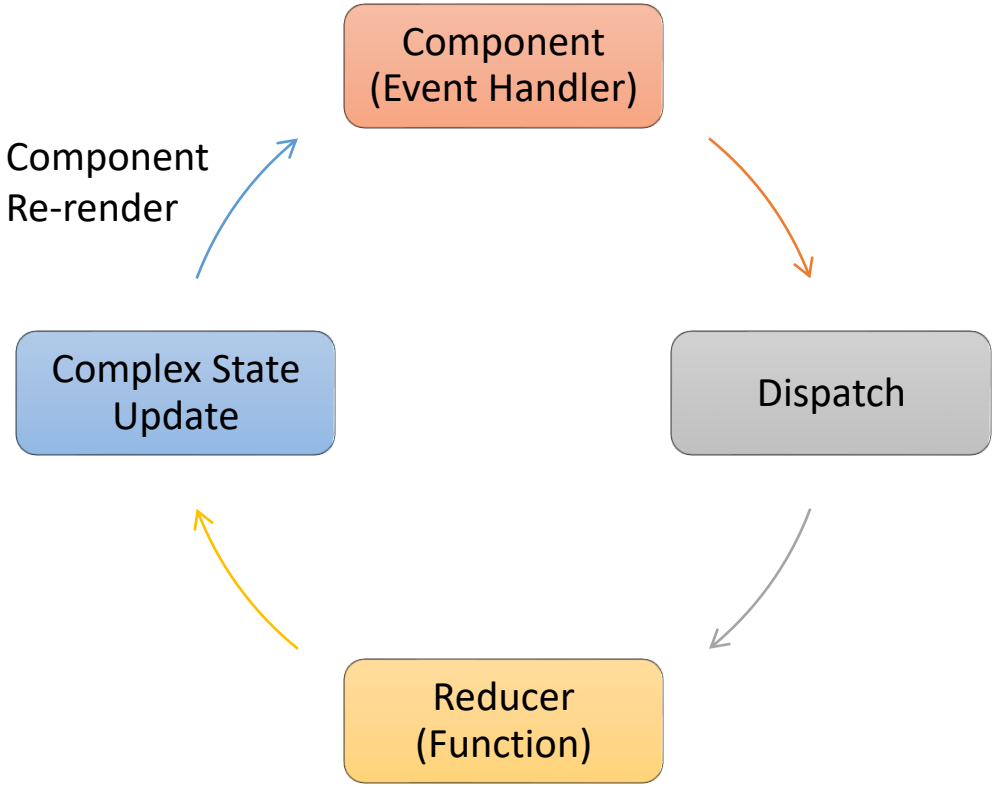
oks > JS UseReducer.js > ...

```jsx
// Reducer function
const fnReducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};

export default UseReducer;
```

Q. What is useReducer() hook? When to use useState() and when useReducer()? V. IMP.

- useReducer() is an alternative to the useState() hook when dealing with **complex state** in your components.

```
// Structure
const [state, dispatch] = useReducer(reducer, initialState);
```



Q. What are the differences between useState() and useReducer() Hook?

useState	useReducer
1. useState is **simpler** to use and used for managing simple state values.	useReducer is more appropriate for **complex state** logic or when the next state depends on the previous one.
2. It is suitable for managing a **single piece of state**.	It is well-suited for managing **multiple pieces of state** that needs to be updated together.
3. The useState hook takes an initial state as an argument and returns an array with two elements: the current state and a function to update the state.	The useReducer hook takes a reducer function and an initial state as arguments and returns the current state and a dispatch function.

Q. What are dispatch & reducer function in useReducer Hook?

- dispatch function is returned by the useReducer hook, and it is used to dispatch actions(type) to reducer function that trigger state updates.
- The reducer function is a function responsible for updating the state based on the action received from dispatch function.

```jsx
// Reducer function
const fnReducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};
```

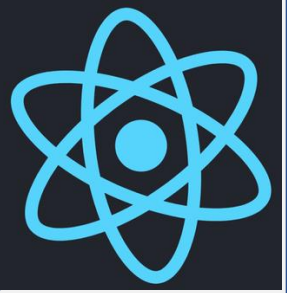
```jsx
// useReducer for managing complex state
import React, { useReducer } from "react";

const UseReducer = () => {
  const i = { count: 0 }; // "i" is object
  const [state, dispatch] = useReducer(fnReducer, i);

  const increment = () => {
    dispatch({ type: "INCREMENT" });
  };
  const decrement = () => {
    dispatch({ type: "DECREMENT" });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

Q. What is the purpose of passing initial state as an object in UseReducer?

- By setting initial state as an object, you can manage complex states with useReducer functions.

```
const initialState = {
  user: {
    name: 'John',
    age: 30,
  },
  isLoggedIn: false,
};
const [state, dispatch] = useReducer(reducer, initialState);
```

9: Hooks - useCallback/ useMemo/ useRef/ useLayoutEffect



Q1. What is the role of useCallback() hook in React? **V. IMP.**

Q2. What parameters does the useCallback hook accept & what does it returns?

Q3. What is the role of useMemo() Hook? **V. IMP.**

Q4. What is the role of useRef() Hook?

Q5. What is useLayoutEffect() Hook? Compare it with useEffect() hook.

Q6. When to use useLayoutEffect() Hook? **V. IMP.**

Q. What is the role of useCallback() hook in React? V. IMP.

- The useCallback hook is used to **memorize functions** in functional components.
- When passing function as a prop from parent component to child component, then passing memorized function avoid unnecessary re-creation of the function on each render.

Count: 0

Increment

oks > JS UseCallbackChild.js > ...

```jsx
const UseCallbackChild = ({ onIncrement }) => {
  return (
    <div>
      <button onClick={onIncrement}>
        Increment</button>
    </div>
  );
};
```

oks > JS UseCallbackParent.js > ...

```jsx
const UseCallbackParent = () => {
  const [count, setCount] = useState(0);

  // Without useCallback
  // const handleIncrement = () => {
  //   setCount(count + 1);
  // };

  // With useCallback
  const handleIncrement = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      {/* Pass the memoized callback to the ChildComponent
      <UseCallbackChild onIncrement={handleIncrement} />
    </div>
  );
};
```

Q. What parameters does the useCallback hook accept & what does it returns?

```
const memoizedCallback = useCallback(callback, dependencies);
```

- The useCallback hook returns the memoized version of the callback function.
- The first parameter is the callback function that you want to memorize.
- The second parameter is an array of dependencies. It specifies the values that, when changed, will cause the memoized callback to be re-created.

Q. What is the role of useMemo() Hook? V. IMP.

- The useMemo hook is a performance optimization function that **memorizes the result** of a computation.
- The advantage is it **prevent unnecessary recalculations and rendering**. It is particularly useful when dealing with expensive calculations.
- userMemo accept two parameters:
    1. The first argument of useMemo() is a function that performs some expensive computation and return some result(processData).
    2. The second argument([data]) is an array of dependencies ([data]). The memorized value will only be recalculated if this dependency will change.
- useMemo returns a memorized version of the calculated value.

oks > JS UseMemo.js > ...

```jsx
import React, { useMemo } from 'react';

const UseMemo = ({ data }) => {
  const processedData = useMemo(() => {
    // Expensive computation on data
    return processData(data);
  }, [data]);

  return (
    <div>
      {/* Render using the memoized value */}
      {processedData}
    </div>
  );
};

export default UseMemo;
```

Q. What is the role of useRef() Hook? V. IMP.

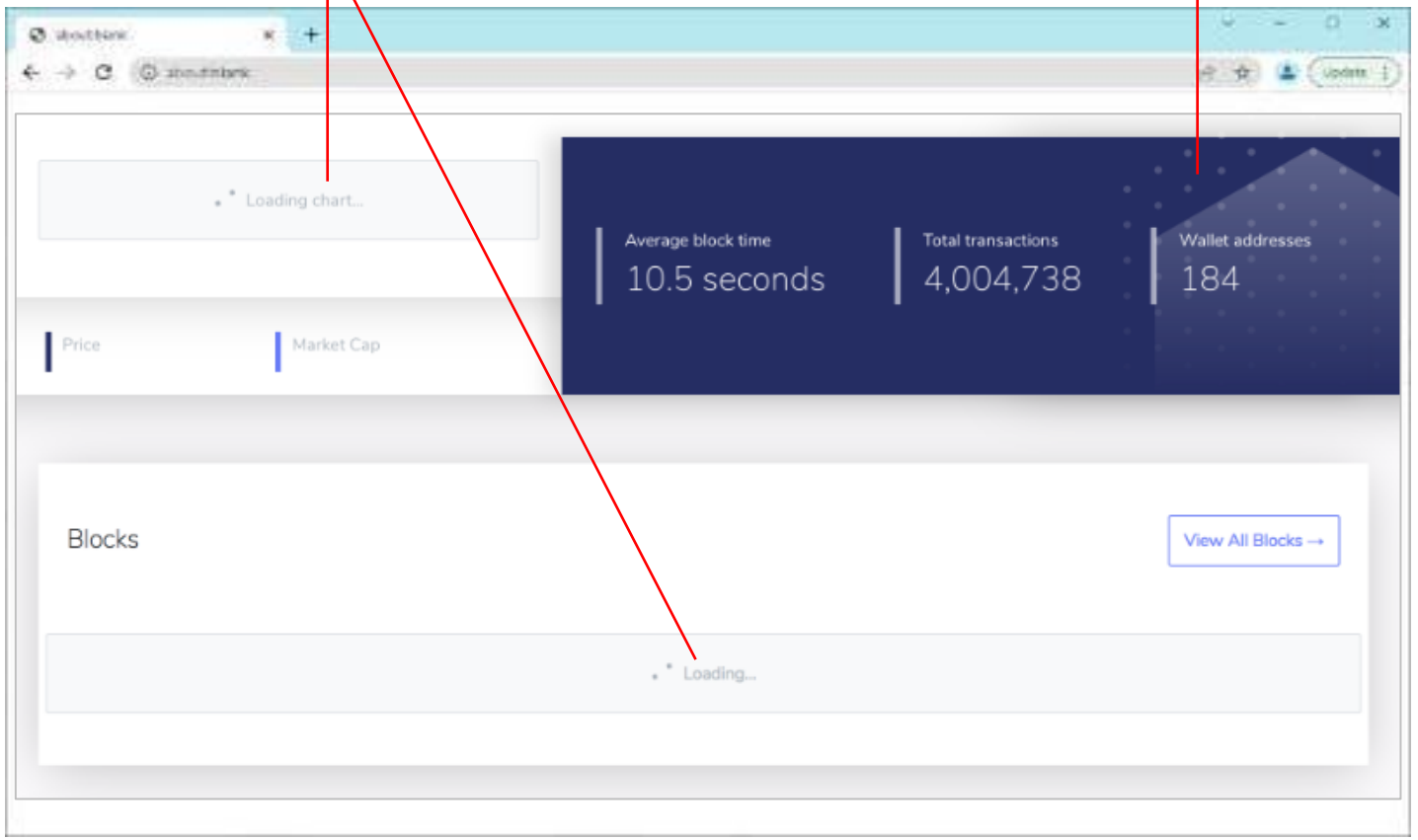
- The useRef hook is primarily used for accessing and **interacting with the DOM elements**.
- The advantage is, the useRef hook **persists across renders** and does not trigger re-renders when it changes.


```jsx
import React, { useRef } from "react";

const UseRef = () => {
  // Create a ref to hold a reference to the element
  const inputRef = useRef();

  // Function to focus the input element
  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />

      {/* Button click will focus on input element */}
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
};
export default UseRef;
```

Q. What is useLayoutEffect() Hook? Compare it with useEffect() hook.

Rendered after side effects by useEffect(eg: Loading data from external API)

Rendered initially



Q. What useLayoutEffect() Hook? Compare it with useEffect() hook.

ooks > JS UseEffect.js > ...

```
function UseEffect() {
  const [data, setData] = useState({});

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      const result = await response.json();
      setData(result);
    };
    fetchData();
  }, []);

  return (
    <div>
      <p>Title: {data.title}</p>
    </div>
  );
}
```

oks > JS UseLayoutEffect.js > ...

```
function UseLayoutEffect() {
  const [data, setData] = useState({});

  useLayoutEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      const result = await response.json();
      setData(result);
    };
    fetchData();
  }, []);

  return (
    <div>
      <p>Title: {data.title}</p>
    </div>
  );
}
```

Same Code

Q. What useLayoutEffect() Hook? Compare it with useEffect() hook.

useEffect

useEffect runs asynchronously and is scheduled after the UI has been rendered.

This means useEffect won't block the browser from updating the UI.

useLayoutEffect

useLayoutEffect run synchronously with the UI rendering.

This means useLayoutEffect can block the browser from updating the UI.

Q. When to use useLayoutEffect() Hook? V. IMP.

- For example, below the paragraph element style(bold/ normal) is dependent on the side effect data, then we use useLayoutEffect.

Update this element to bold after data fetch

sunt aut facere repellat provident occaecati excepturi optio reprehenderit

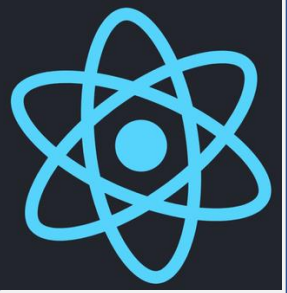
```
  return (
    <div>
      <p id="myElement">Update this element
      to bold after data fetch</p>

      {data && <p>{data.title}</p>}
    </div>
  );
};

export default UseLayoutEffectUse;
```

```
const UseLayoutEffectUse = () => {
  const [data, setData] = useState(null);

  useLayoutEffect(() => {
    const fetchData = async () => {
      // Fetch API data
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      const result = await response.json();
      setData(result);
      // Update styles based on the fetched data
      const element = document.getElementById("myElement");
      if (element && result && result.title) {
        element.style.fontWeight = "bold";
      }
    };

    fetchData();
  }, []);
```

Q. When to use useLayoutEffect() Hook? V. IMP.

- The useLayoutEffect hook in React should be used when you need to perform actions that involve manipulating the layout of the DOM based on side effects, such as data fetching.
- In other words, you want the layout changes to be reflected before the browser paints the updated UI.

10: Hooks - Best Practices/ Short Answer



Q1. What are the Rules or Best Practices for hooks implementation? **V. IMP.**

Q2. What are Custom Hooks?

Q3. Tell 3 scenarios in which you can use useEffect() hook?

Q4. What useState() hook return in React?

Q5. How can you conditionally run effects with useEffect?

Q6. What problem do React Hooks solve?

Q7. What is the advantage of React Hooks over Lifecycle methods?

Q8. What is Context API? **V. IMP.**

Q9. What are the uses of all the Hooks in React? **V. IMP.**

Q. What are the Rules or Best Practices for hooks implementation? V. IMP.

❖ **Rules or Best Practices for hooks implementation**

1. Hooks can only be called at the top level of a component.
2. Hooks can only be called directly inside React components.
3. Keep hooks order consistent.

```
function MyComponentRule1() {
  // Use Hooks at the Top Level
  const [count, setCount] = useState(0);
  // Remaining code of component...
}

function MyComponent2Rule1() {
  // Component code...
  // Hook at middle or bottom
  const [count, setCount] = useState(0);
}
```

```
function MyComponentRule2(){
  // Only call Hooks from React Components
  const [count, setCount] = useState(0);
}
function someFunction() {
    // Not from normal or nested JS functions
    const [count, setCount] = useState(0);
}
```

```
function MyComponent1Rule3(){
    // Keep Hooks order consistent
    const [count, setCount] = useState(0);
    const [text, setText] = useState('');
}
function MyComponent2Rule3(){
    // Don't change hook order
    const [text, setText] = useState('');
    const [count, setCount] = useState(0);
}
```

Q. What are Custom Hooks?

- Custom Hooks in React are JavaScript functions developed by developers to encapsulate reusable logic for their applications.

oks > JS useCustomHooks.js > ...

```
// Custom Hook for managing a counter
const useCustomHook = (initialValue = 0) => {
  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return { count, increment, decrement };
};
export default useCustomHook;
```

oks > JS UseCustomHookUse.js > ...

```
const UseCustomHookUse = () => {
  const { count, increment, decrement } = useCustomHook();

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default UseCustomHookUse;
```

Q. What useState() hook return in React?

1. StateVariable is the current state value.

1. stateUpdaterFunction is a function that used to update the state.

```
const [stateVariable, stateUpdaterFunction] = useState(initialState);
```

Q. Tell 3 scenarios in which you can use useEffect() hook?

useEffect() hook can be used for:

1. Fetching data from external API.
2. For managing subscriptions or event listeners.
3. For manual DOM manipulations.
4. For setting up timers or intervals.

Q. How can you conditionally run effects with useEffect?

- In React, you can conditionally run effects with the useEffect hook by **placing the if-else condition inside** the hook itself.

```jsx
const UseEffectCondition = () => {
  const [shouldRunEffect, setShouldRunEffect] = useState(true);

  useEffect(() => {
    // Conditionally run the effect
    if (shouldRunEffect) {
      // Your effect code here
      console.log("Effect ran!");
    }
  }, [shouldRunEffect]); // Include any dependencies if needed

  return (
    <div>
      <p>Conditionally Run Effect Example</p>
      <button onClick={() => setShouldRunEffect(!shouldRunEffect)}>
        Toggle Effect
      </button>
    </div>
  );
};
export default UseEffectCondition;
```

Q. What problem do React Hooks solve?

- React Hooks solve problems related to managing **state**, **lifecycle**, and **side effects** in functional components

Q. What is the advantage of React Hooks over Lifecycle methods?

- **Simplicity:** Hooks eliminate the need for class components, making the code more readable and reducing boilerplate.

Q. What is Context API?

- Context API is a **broader concept** for the creation of context and its management.
- It can be achieved by **using useContect Hook**.

ooks > JS MyContext.js > ...

```
import { createContext } from "react";

const MyContext = createContext();

export default MyContext;
```

Hello from Context!

ooks > JS Parent.js > ...

```
const Parent = () => {
  const contextValue = "Hello from Context!";

  return (
    <MyContext.Provider value={contextValue}>
      {/* Your component tree */}
      <Child></Child>
    </MyContext.Provider>
  );
};
export default Parent;
```

ooks > JS Child.js > ...

```
const Child = () => {
  const contextValue = useContext(MyContext);

  return <p>{contextValue}</p>;
  // return (
  //   <MyContext.Consumer>
  //     {(contextValue) => <div>{contextValue}</div>}
  //   </MyContext.Consumer>
  // );
};
export default Child;
```

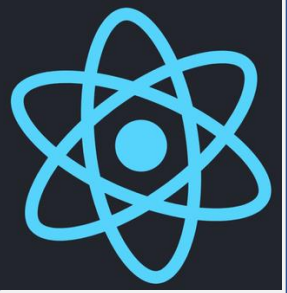
Q. What are the uses of all the Hooks in React? V. IMP.

1. useState:	State
2. useEffect:	Side effects
3. useContext:	Context
4. useReducer:	Complex state
5. useCallback:	Memorize function
6. useMemo:	Memorize function result
7. useRef:	Refs
8. useLayoutEffect:	Synchronous Side effects.

Q. What are the uses of all the Hooks in React? V. IMP.

1. useState:	Used for adding state to functional components.
2. useEffect:	Enables performing side effects in functional components, such as data fetching.
3. useContext:	Allows functional components to consume values from a React context.
4. useReducer:	Provides an alternative to **useState** when the state logic is more complex.
5. useCallback:	Memoizes a callback function to prevent unnecessary re-renders of child components.
6. useMemo:	Memoizes the result of a function to optimize performance by avoiding unnecessary recalculations.
7. useRef:	Creates a mutable object that persists across renders (used for interacting with DOM elements).
8. useLayoutEffect:	Similar to **useEffect** but fires synchronously after all DOM mutations.

11: Components LifeCycle Methods - I



Q1. What are Component life cycle phases? **V. IMP.**

Q2. What are Component life cycle methods? **V. IMP.**

Q3. What are Constructors in class components? When to use them?

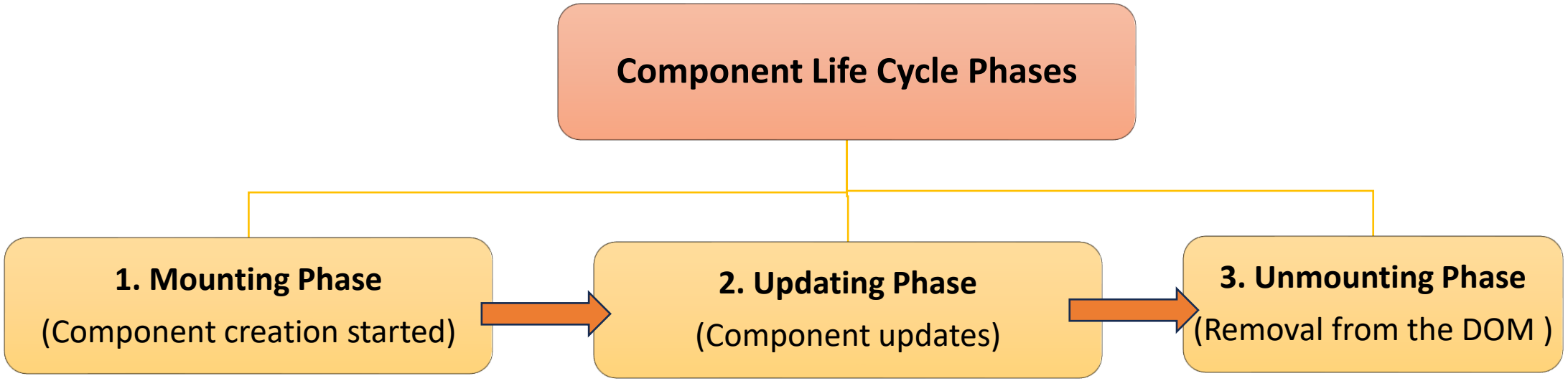
Q4. What is the role of super keyword in constructor?

Q5. What is the role of render() method in component life cycle?

Q6. How the State can be maintained in a class component?

Q7. What is the role of componentDidMount() method in component life cycle?

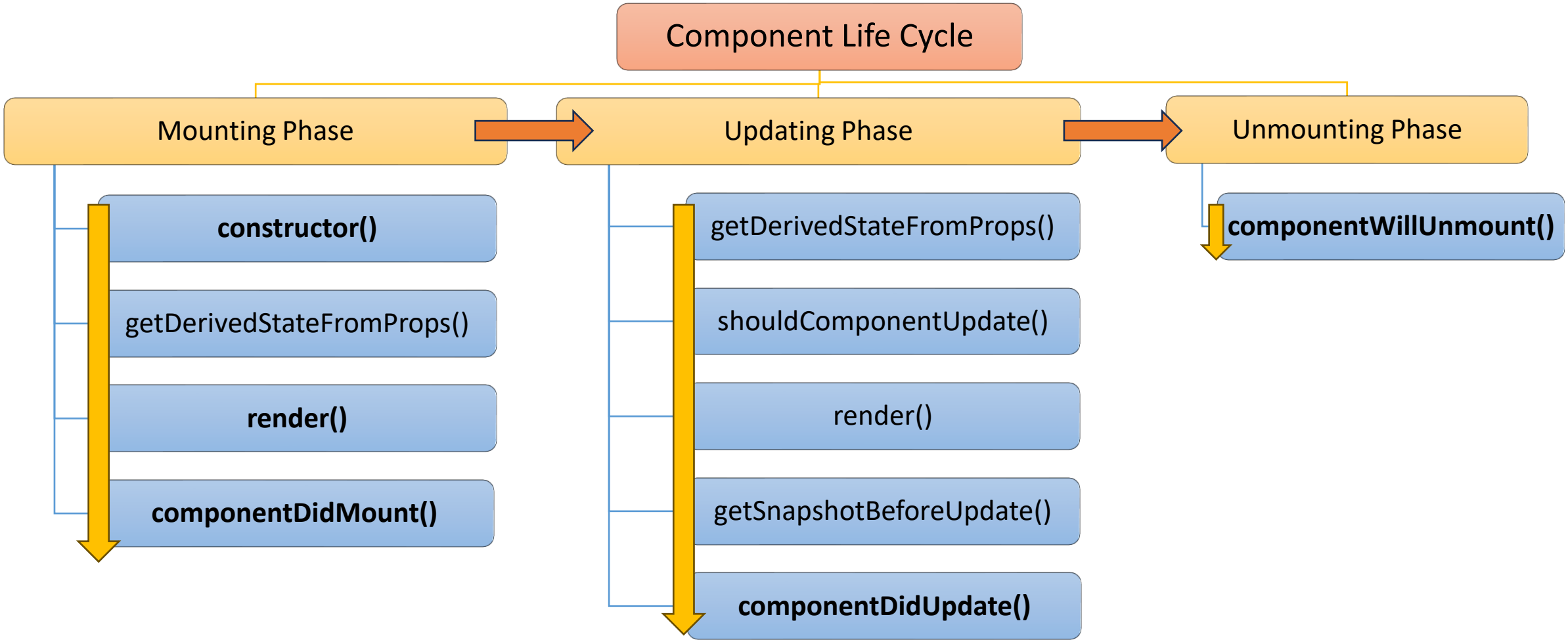
Q. What are Component life cycle phases?



- This phase occurs when an instance of a component is being created and inserted into the DOM.
- This phase occurs when a component is being re-rendered as a result of changes to either its props or state.
- This phase occurs when a component is being removed from the DOM.

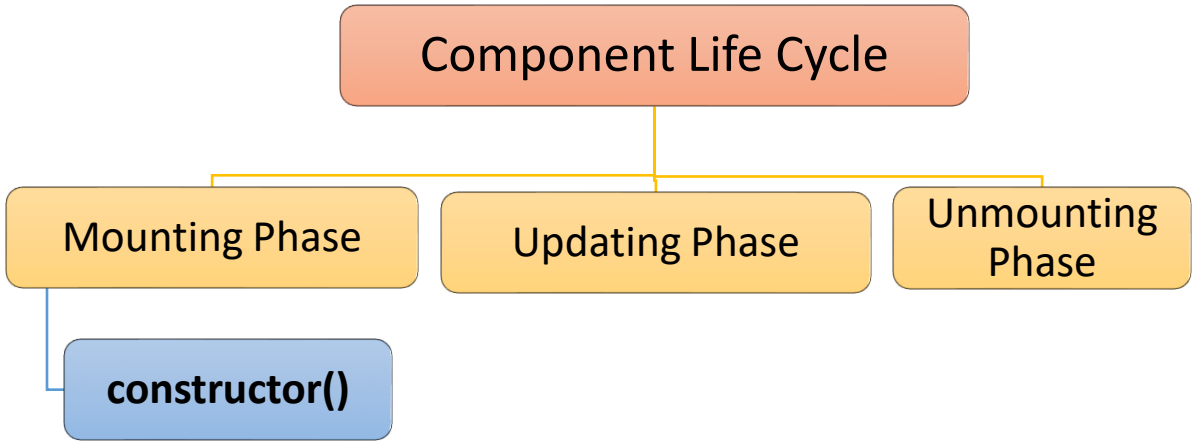
Q. What are Component life cycle methods? V. IMP.

- Component lifecycle methods are special methods that get called at various stages of a component's life.



Q. What are Constructors in class components? When to use them?

- constructor is a **special method** that is called when an instance of the class is created.
- Constructor is **used for initializing the component's state** or performing any setup that is needed before the component is rendered.



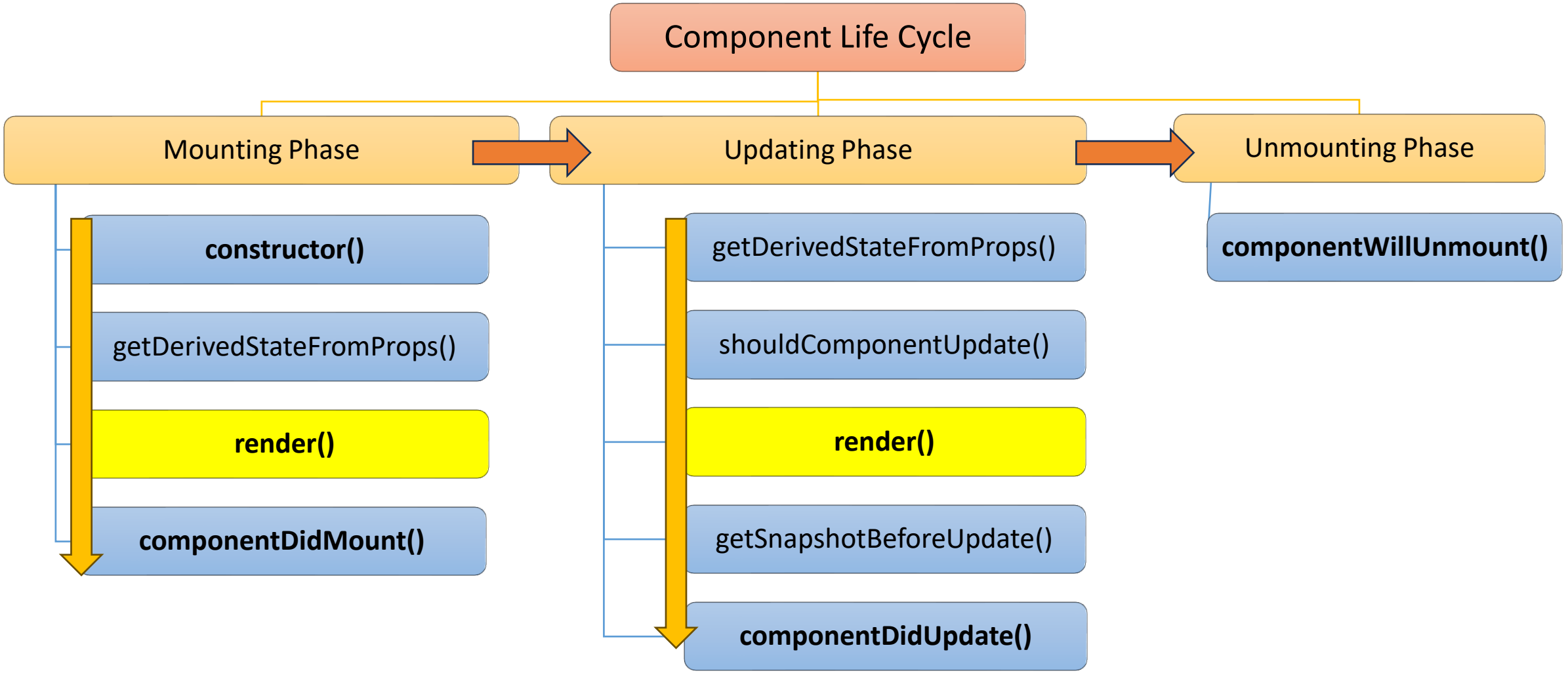

```
class ConstructorExample extends Component {
  constructor(props) {
    super(props);

    // Initialize the state
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
        <h2>Count: {this.state.count}</h2>
    );
  }
}

export default ConstructorExample;
```

Q. What is the role of super keyword in constructor?

- super keyword is used in the constructor of a class component to **call the constructor of the parent class**.
- This is necessary to ensure that the initialization logic of the parent class is executed.

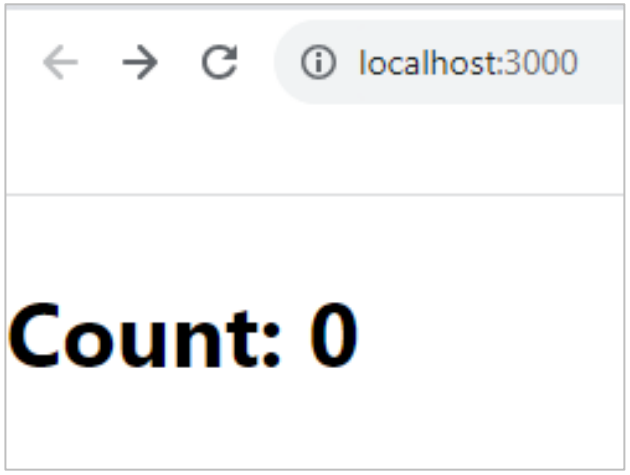
```
class ConstructorExample extends Component {
  constructor(props) {
    super(props);

    // Initialize the state
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
        <h2>Count: {this.state.count}</h2>
    );
  }
}

export default ConstructorExample;
```

Q. What is the role of render() method in component life cycle?



Q. What is the role of render() method in component life cycle?

- Render() method **returns the React elements** that will be rendered to the DOM.




```
class ConstructorExample extends Component {
  constructor(props) {
    super(props);

    // Initialize the state
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
      <h2>Count: {this.state.count}</h2>
    );
  }
}

export default ConstructorExample;
```

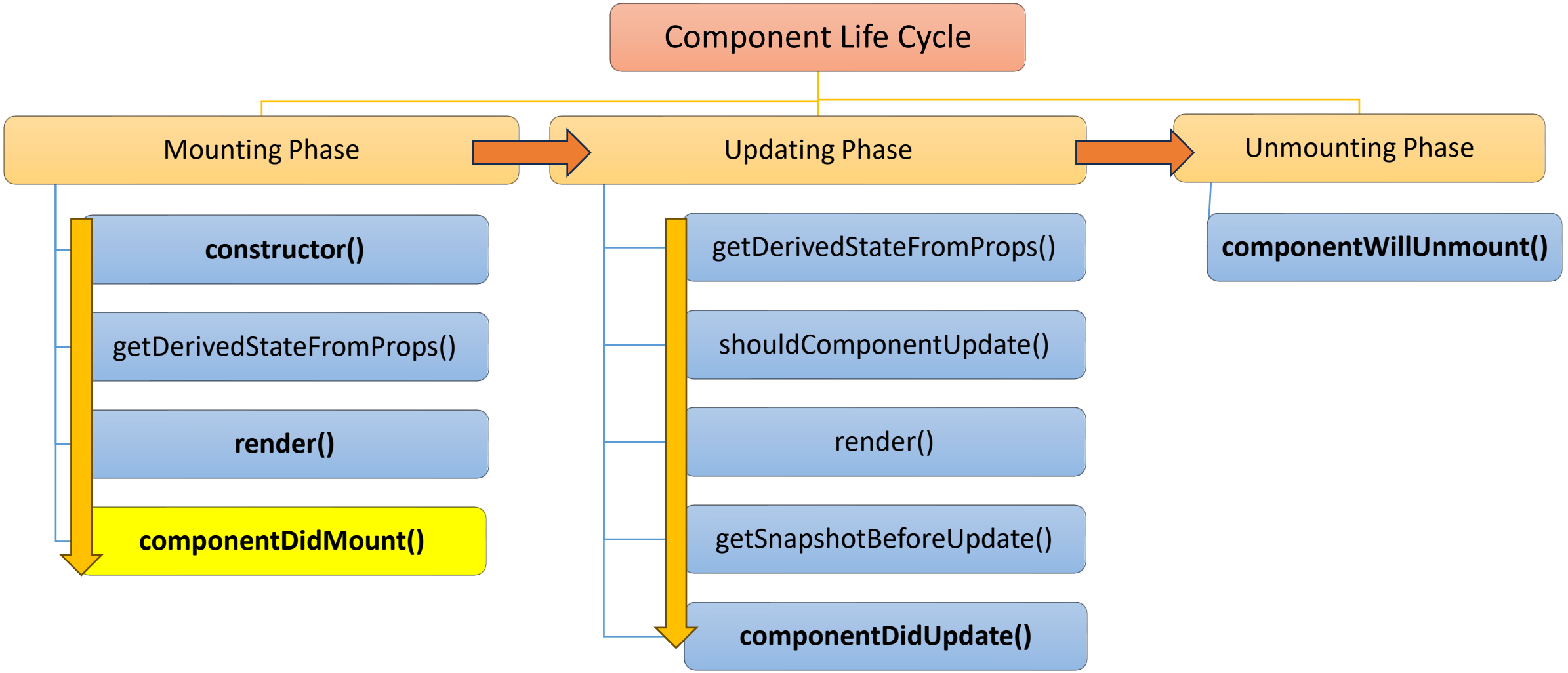
Q. How the State can be maintained in a class component? V. IMP.

❖ **Two step process to maintain state:**

1. **this.setState()** method is used to update the state.
2. **this.state** property is used to render the update state in DOM.


```
class StateComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0, // Initializing the state
    };
  }
  render() {
    return (
      <div>
        <h2>Counter: {this.state.count}</h2>
        <button onClick={this.handleInc}>Inc</button>
      </div>
    );
  }

  handleInc = () => {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  };
}
export default StateComponent;
```

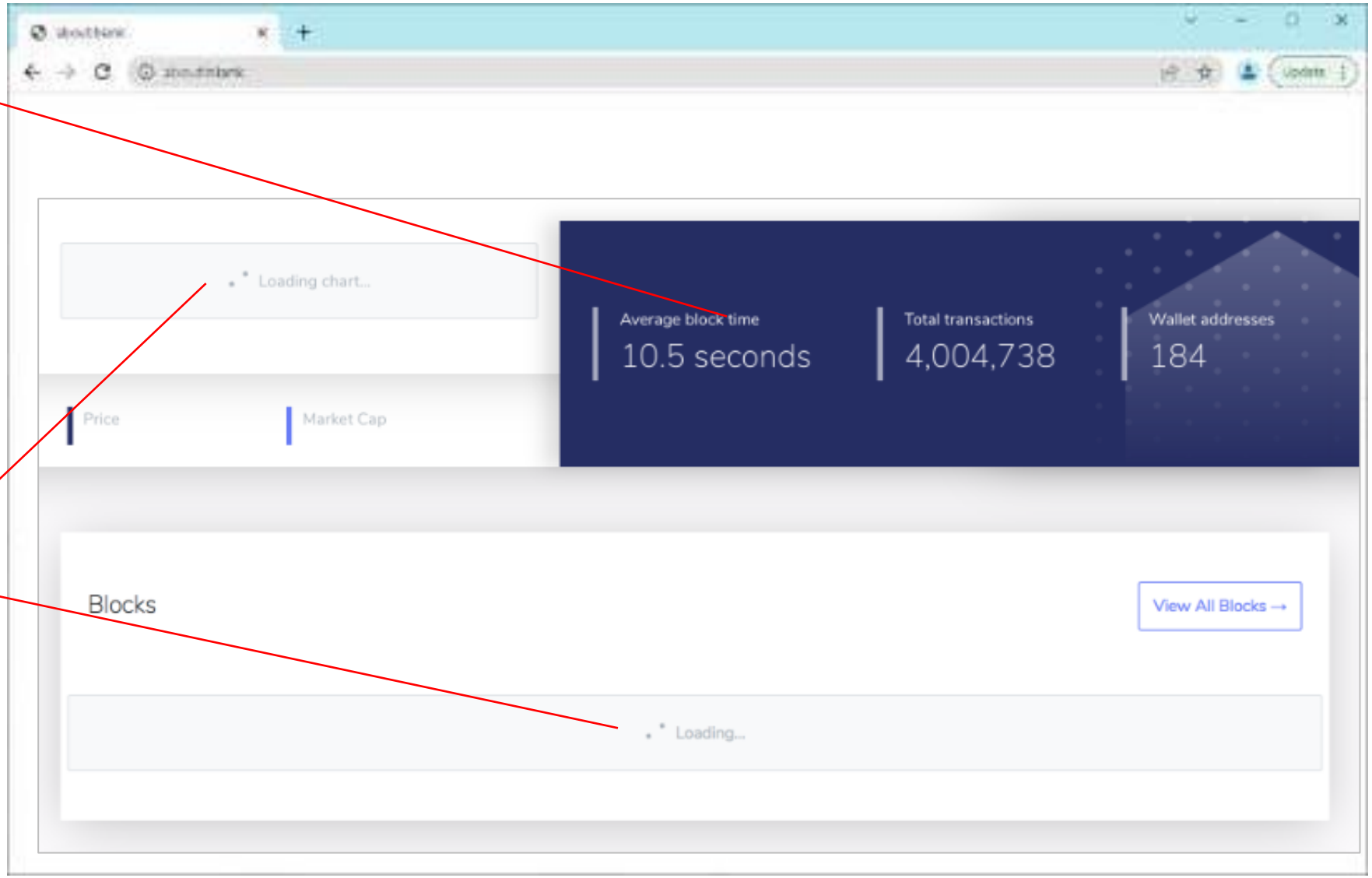
Q. What is the role of componentDidMount() method in component life cycle?



Q. What is the role of componentDidMount() method in component life cycle?

1. Rendered after constructor initialization.

2. Rendered after componentDidMount() to run **Side effects**(ex: loading data from external API) and then call render() method of updating phase again.

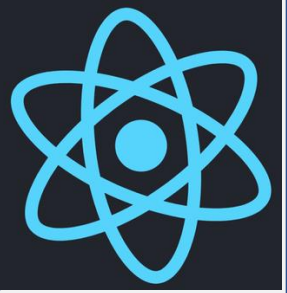


Q. What is the role of componentDidMount() method in component life cycle?

- componentDidMount() lifecycle method in React is the part of mounting phase and is **called after a component has been rendered** to the DOM.
- Mostly used for **side effects**. For example, external data fetching or setting up subscriptions.

```
// Fetch data from API example using
// componentDidMount life cycle method
class CompDidMount extends Component {
  constructor(props) {
    // 1. Component intialization
    this.state = {
      data: null
    };
  }

  // 2. componentDidMount is called after
  // the component is added to the DOM
  componentDidMount() {
    // 3. Fetch data from an API and
    // update state with fetch data
    fetchData().then((data) => {
      this.setState({
        data: data
      });
    });
  }

  // 4. Render the component's UI using
  // the fetched data
  render() {
    return (
      <div>
        <p>Data: {this.state.data}</p>
      </div>
    );
  }
}
export default CompDidMount;
```

12: Components LifeCycle Methods - II



Q1. What is the role of componentDidUpdate() method in component life cycle?

Q2. What is the role of componentWillUnmount() method in component life cycle?

Q3. How do you initialize state in a class component?

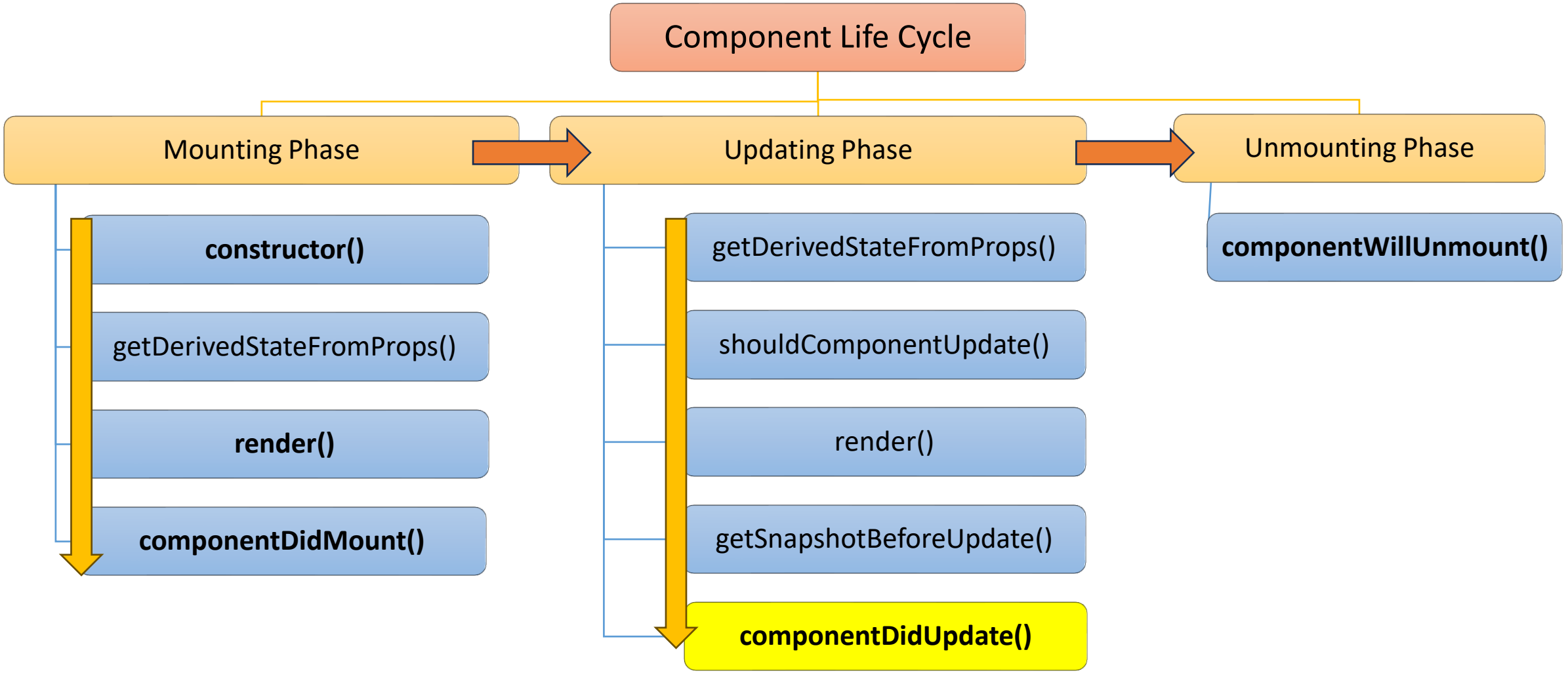
Q4. In which lifecycle phase component will be re-rendered?

Q5. What will happen if you don’t define a constructor in your React component?

Q6. Why we need class components when we already have functional components?

Q7. What are the 5 main methods of component lifecycle?

Q. What is the role of componentDidUpdate() method in component life cycle?

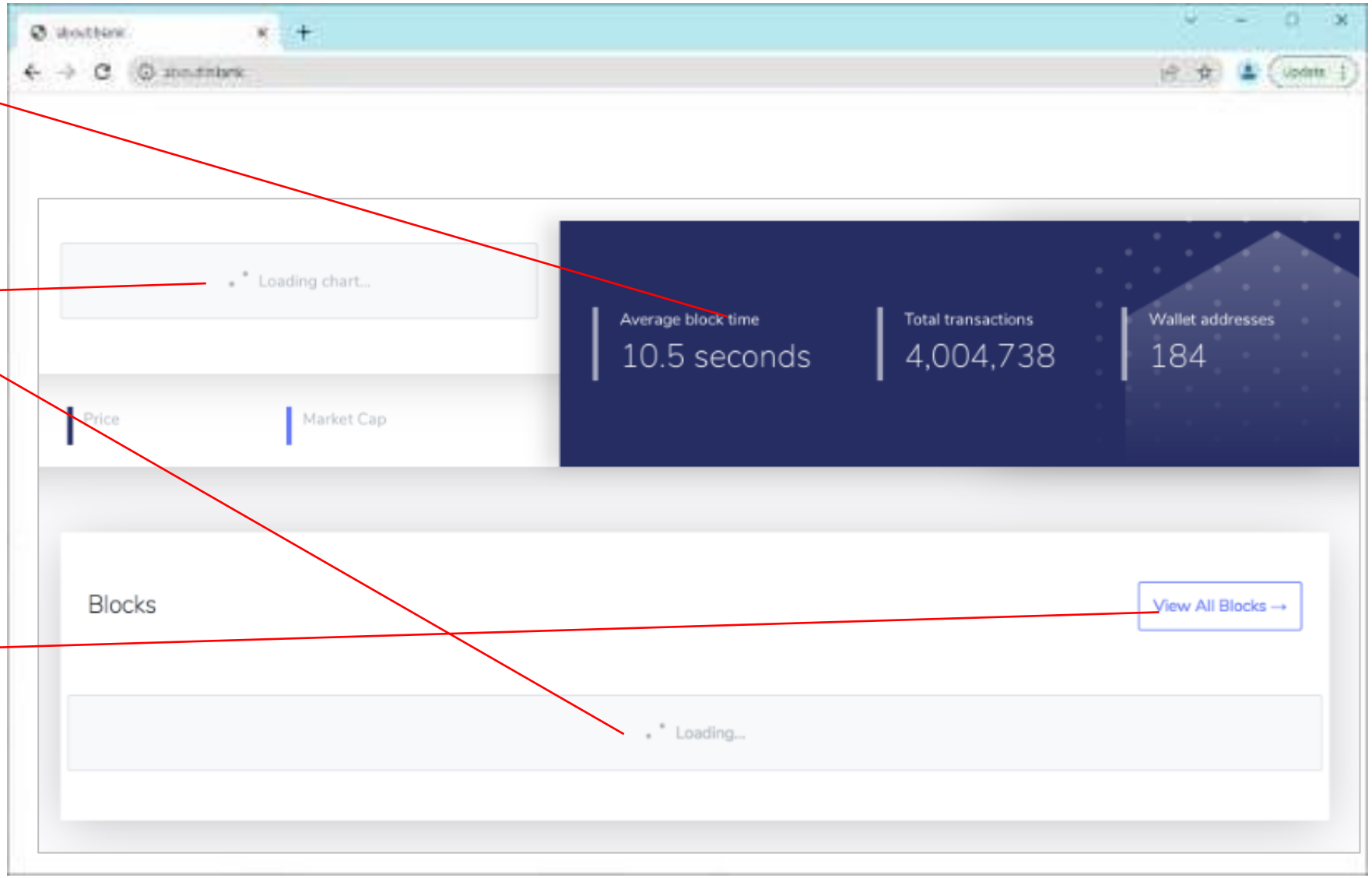


Q. What is the role of componentDidUpdate() method in component life cycle?

1. Rendered after constructor initialization.

2. Rendered after componentDidMount() Side effects.

3. For **any property or state change**, componentDidUpdate() method will refetch the data and re-rendered in the DOM.



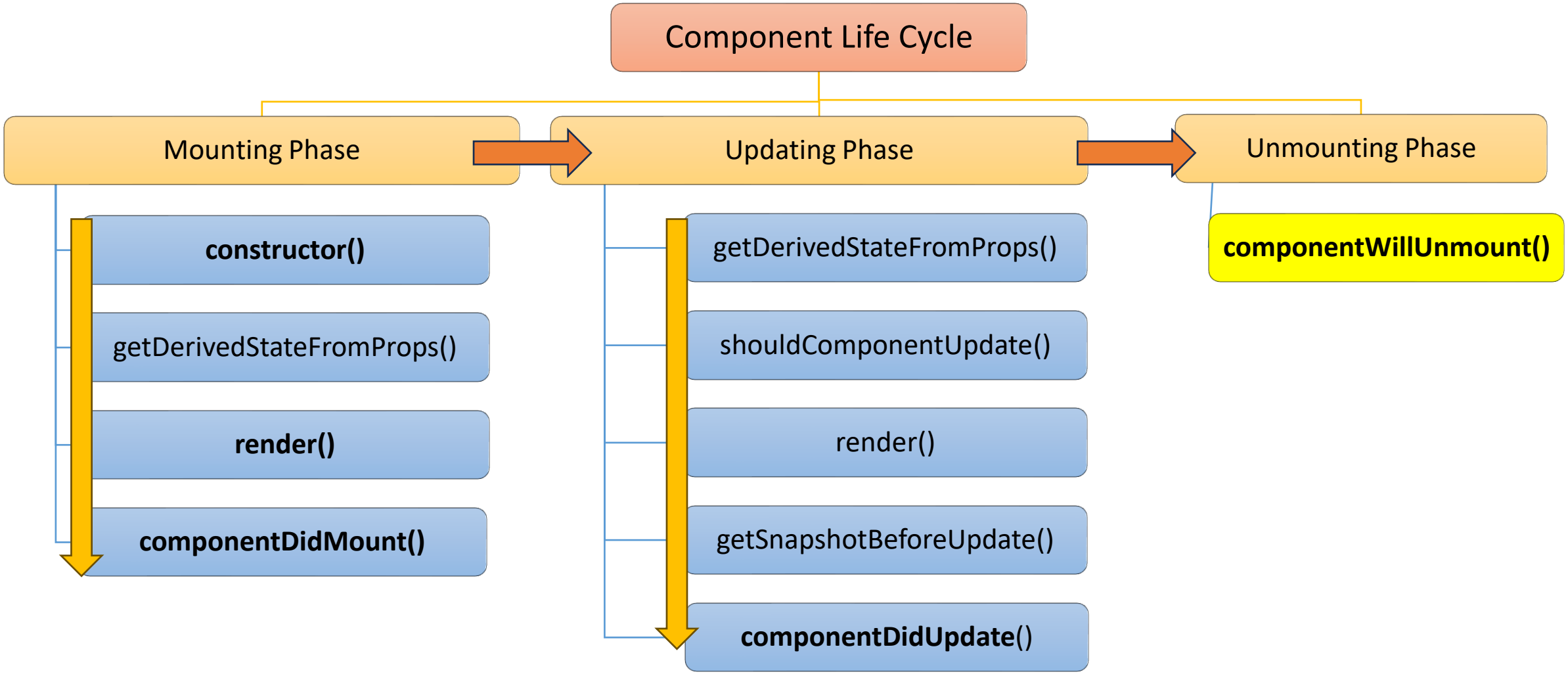
Q. What is the role of componentDidUpdate() method in component life cycle?

- componentDidUpdate() lifecycle method is the part of updating phase and is called **after a component has been updated**(property of state change).

- For example, after data is initially fetched by the componentDidMount method, if the user changes some property in the component (e.g., triggering a re-render with updated props or state), then the componentDidUpdate method can be used to re-fetch data again.

```
class CompDidUpdate extends Component {
  constructor(props) {
    this.state = { data: null,};
  }
  componentDidMount() {
    fetchData().then((data) => {
      this.setState({ data: data });
    });
  }

  componentDidUpdate(prevProps) {
    // Fetch new data if userId property changes
    if (this.props.userId !== prevProps.userId) {
      fetchData().then((data) => {
        this.setState({ data: data });
      });
    }
  }
  render() {
    return <div>Data: {this.state.data}</div>;
  }
}
export default CompDidUpdate;
```

Q. What is the role of componentWillUnmount() method in component life cycle?



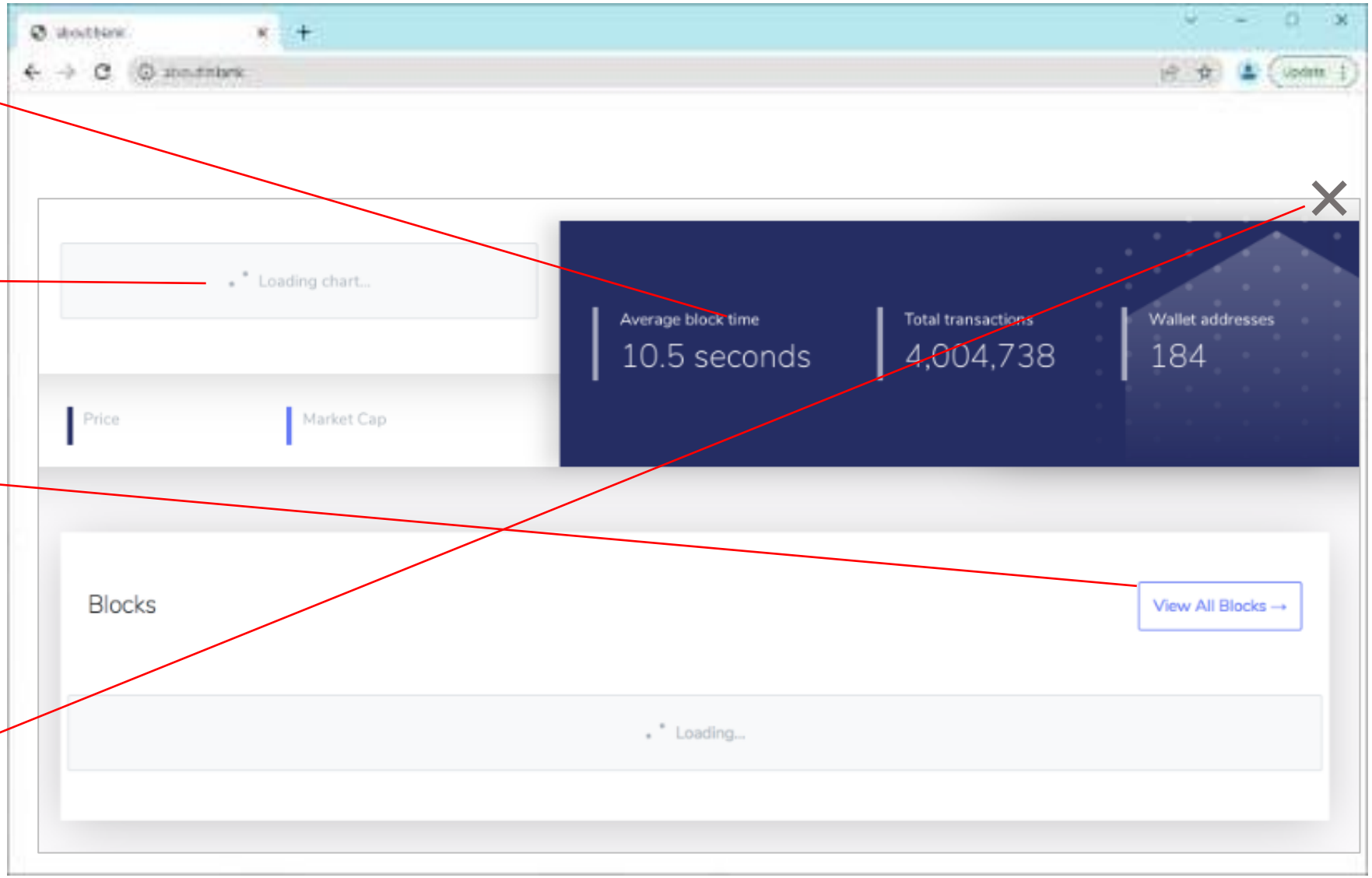
Q. What is the role of componentWillUnmount() method in component life cycle?

1. Rendered after **constructor()** initialization.

2. Rendered after **componentDidMount().**

3. Re-rendered after, **componentDidUpdate()** on any state or property change.

3. Before unmounting of component, **componentWillUnmount()** will do the cleanup task.



Q. What is the role of componentWillUnmount() method in component life cycle?

- componentWillUnmount() lifecycle method is called just **before a component is unmounted** and removed from the DOM.

- It is used for **cleanup tasks**, freeing up resources or canceling any ongoing processes. For example, remove event listener or unsubscribe from any external service.

```
// pseudocode: componentWillUnmount
class MyComponent extends Component {
  constructor(props) {
    // Intialize component...
  }

  componentDidMount() {
    // Add event listner
    document.addEventListener("click", this.handleClick);
  }

  componentWillUnmount() {
    // 1. Clean up operations
    document.removeEventListener("click", this.handleClick);

    // 2. Unsubscribe or close connections
    externalService.unsubscribe(this.subscription);
  }
}
```

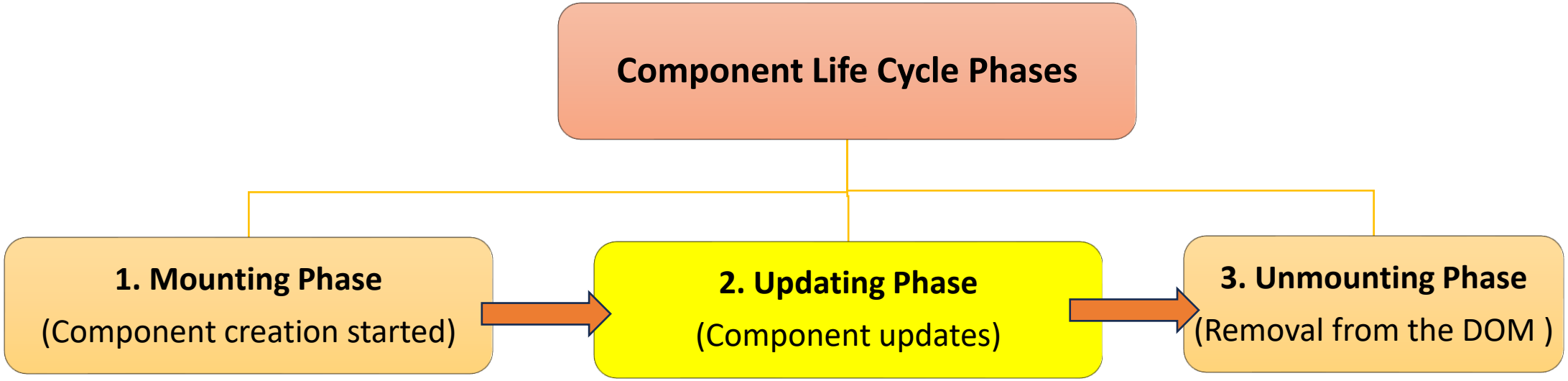
Q. How do you initialize state in a class component?

- In a React class component, state is initialized in the **constructor**.

```
class ConstructorExample extends Component {
  constructor(props) {
    super(props);

    // Initialize the state
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
        <h2>Count: {this.state.count}</h2>
    );
  }
}

export default ConstructorExample;
```

Q. In which lifecycle phase component will be re-rendered?

- In the **updating phase** component will be re-rendered if any props or state is changed.



Q. What will happen if you don’t define a constructor in your React component?

- If you don't define a constructor in your React component, React will **automatically create a default constructor for you**. This default constructor will call the constructor of the base class (Component) using super(props).

```
class ConstructorExample extends Component {

  render() {
    return (
      <h2>Interview Happy</h2>
    );
  }
}

export default ConstructorExample;
```

Q. Why we need class components when we already have functional components?

- Previously, class components were used for managing state in React applications since functional components were stateless. However, with the introduction of hooks, functional components can also handle the state.

- When we need class components now?

1. To manage **existing projects(legacy code)**, especially those built before the introduction of hooks.
2. To support some **third-party libraries** that might still be written as class components.
3. To support component lifecycle methods of the existing legacy code.

Q. What are the 5 main methods of component lifecycle?

Component Life Cycle Main Methods

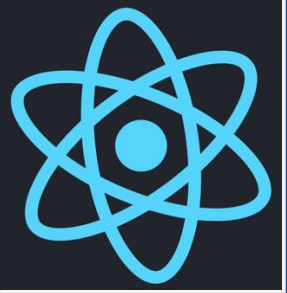
1. constructor()
2. render()
3. componentDidMount()
4. componentDidUpdate()
5. componentWillUnmount()

```
// pseudocode: summary
class CompleteClassComponent extends Component {
  constructor(props) {
    // 1. Initialize the component
  }

  render() {
    // 2. Render the component
  }

  componentDidMount() {
    // 3. Component is mounted to the DOM
  }

  componentDidUpdate(prevProps, prevState) {
    // 4. Component is updated in the DOM
  }

  componentWillUnmount() {
    // 5. Component is about to be removed from t
  }
}
```

Q. What are the 5 main methods of component lifecycle?

1. constructor()	Initializes the component's state and automatically called when the component is being created.
2. render()	Returns UI elements based on the current state and props;
3. componentDidMount()	Invoked after the component has been added to the DOM; used for data fetching and subscribing to external data sources.
4. componentDidUpdate()	Invoked after a component's state or props have been updated and it has been re-rendered; used for handling side effects after updates.
5. componentWillUnmount()	Called just before a component is unmounted and removed from the DOM; used for cleanup tasks and releasing resources.

13: Controlled & Uncontrolled Components



Q1. What are Controlled Components in React? **V. IMP.**

Q2. What are the Differences btw Controlled & Uncontrolled Components? **V. IMP.**

Q3. What are characteristics of controlled components?

Q4. What are the advantages of using controlled components in React forms?

Q5. How to handle forms in React?

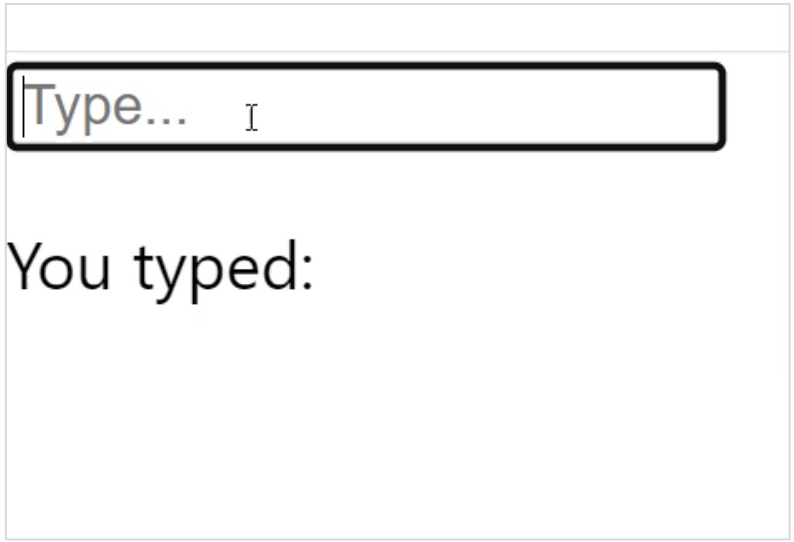
Q6. How can you handle multiple input fields in a controlled form?

Q7. How do you handle form validation in a controlled component?

Q8. In what scenarios might using uncontrolled components be advantageous?

Q. What are Controlled Components in React? V. IMP.

- A controlled component is a component whose form elements (like input fields or checkboxes) are **controlled by the state** of the application.




```jsx
const Controlled = () => {
  // State to store the input value
  const [inputValue, setInputValue] = useState("");

  // Event handler for input changes
  const handleInputChange = (e) => {
    // Update the state with the new input value
    setInputValue(e.target.value);
  };

  return (
    <div>
      {/* Input controlled by the state*/}
      <input type="text" value={inputValue}
        onChange={handleInputChange} placeholder="Type..."/>

      {/* Display the current state value */}
      <p>You typed: {inputValue}</p>
    </div>
  );
};
export default Controlled;
```

Q. What are the Differences btw Controlled & Uncontrolled Components? V. IMP.

```jsx
const Controlled = () => {
  // State to store the input value
  const [inputValue, setInputValue] = useState("");

  // Event handler for input changes
  const handleInputChange = (e) => {
    // Update the state with the new input value
    setInputValue(e.target.value);
  };

  return (
    <div>
      {/* Input controlled by the state*/}
      <input type="text" value={inputValue}
        onChange={handleInputChange} placeholder="Ty

      {/* Display the current state value */}
      <p>You typed: {inputValue}</p>
    </div>
  );
};
export default Controlled;
```

```jsx
const Uncontrolled = () => {
  // Create a ref to access the input value
  const inputRef = useRef(null);

  const handleClick = () => {
    // Access the input value directly using ref
    const value = inputRef.current.value;
    alert(`You typed: ${value}`);
  };

  return (
    <div>
      {/* Uncontrolled input with ref */}
      <input type="text" ref={inputRef}
      placeholder="Type something..." />

      <button onClick={handleClick}>Click</button>
    </div>
  );
};
export default Uncontrolled;
```

Q. What are the Differences btw Controlled & Uncontrolled Components? V. IMP.

Controlled Components	Uncontrolled Components
1. Values are controlled by **React state**.	Values are not controlled by React state.
2. Event handlers **update React state**.	No explicit state update; values can be accessed **directly from the DOM**.
3. Don't depend on useRef().	Commonly uses **useRef()** to access form element values.
4. **Re-renders** on state changes.	Less re-rendering since values are not directly tied to React state.
5. A **recommended** and standard practice for form handling in React.	Useful in certain scenarios but less commonly considered a best practice.

Q. What are characteristics of controlled components?

❖ **Characteristics of controlled components:**

1. **State Control:** The value of the form element is stored in the component's state.
2. **Event Handling:** Changes to the form element trigger an event (e.g., onChange for input fields).
3. **State Update:** The event handler updates the component's state with the new value of the form element.
4. **Re-rendering:** The component re-renders with the updated state, and the form element reflects the new value.

```
const Controlled = () => {
  // State to store the input value
  const [inputValue, setInputValue] = useState("");

  // Event handler for input changes
  const handleInputChange = (e) => {
    // Update the state with the new input value
    setInputValue(e.target.value);
  };

  return (
    <div>
      {/* Input controlled by the state*/}
      <input type="text" value={inputValue}
        onChange={handleInputChange} placeholder="Type..."/>

      {/* Display the current state value */}
      <p>You typed: {inputValue}</p>
    </div>
  );
};
export default Controlled;
```

Q. What are the advantages of using controlled components in React forms?

❖ Top 3 benefits of using controlled components in React forms:

1. In controlled components, form elements have their values managed by React state, ensuring a **single source of truth**.

2. This approach facilitates predictable and synchronized updates, making it easier to implement features such as form validation, and dynamic rendering, and seamless integration with React's lifecycle methods.

3. Controlled components offer better control and maintainability compared to uncontrolled components, making them the best practice for handling forms in React applications.

Q. How to handle forms in React?

- The preferred and recommended approach for handling forms in React is **by using controlled components**.

Q. How can you handle multiple input fields in a controlled form?

- Maintain **separate state variables** for each input field and update them individually using the **onChange event**.

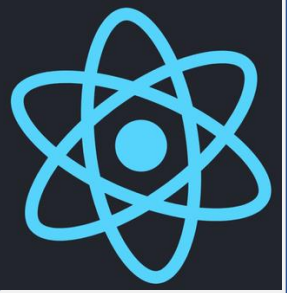
Q. How do you handle form validation in a controlled component?

- By using conditional rendering based on the state and validate input values before updating the state.

Q. In what scenarios might using uncontrolled components be advantageous?

- Uncontrolled components can be beneficial when integrating with non-React libraries, or when dealing with forms where controlled components are not possible.

14: Code Splitting



Q1. What is Code Splitting in React? **V. IMP.**

Q2. How to Implement Code Splitting in React? **V. IMP.**

Q3. What is the role of Lazy and Suspense methods in React? **V. IMP.**

Q4. What are the Pros and Cons of Code Splitting?

Q5. What is the role of the import() function in code splitting?

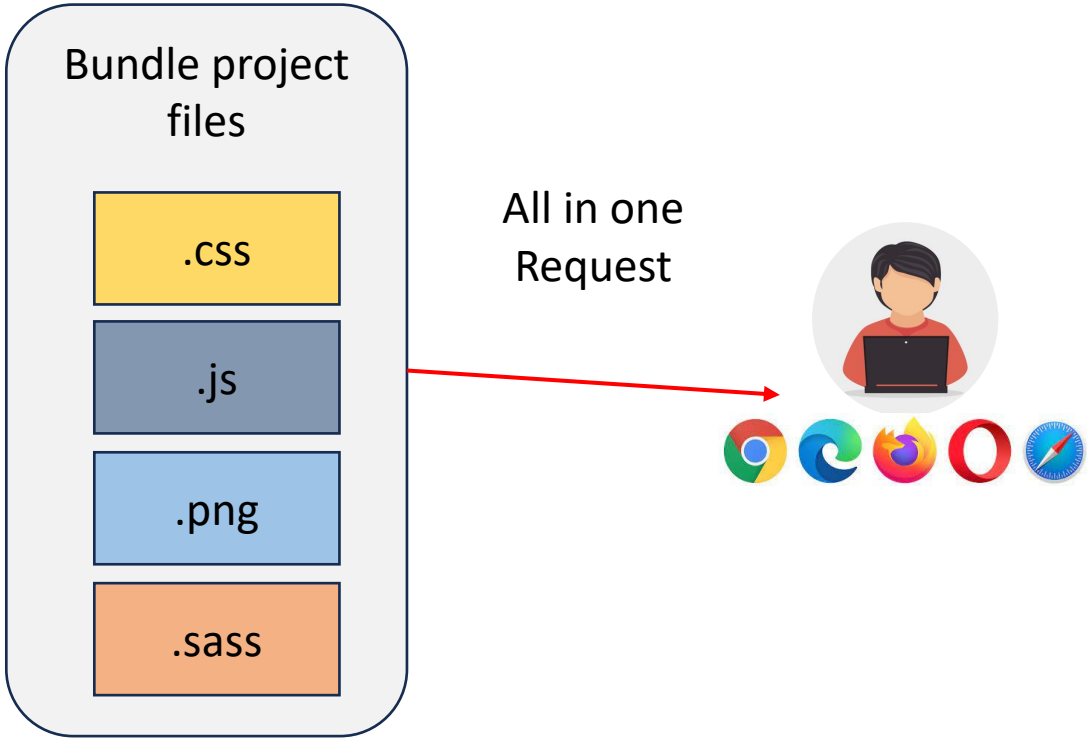
Q6. What is the purpose of the fallback prop in Suspense?

Q7. Can you dynamically load CSS files using code splitting in React?

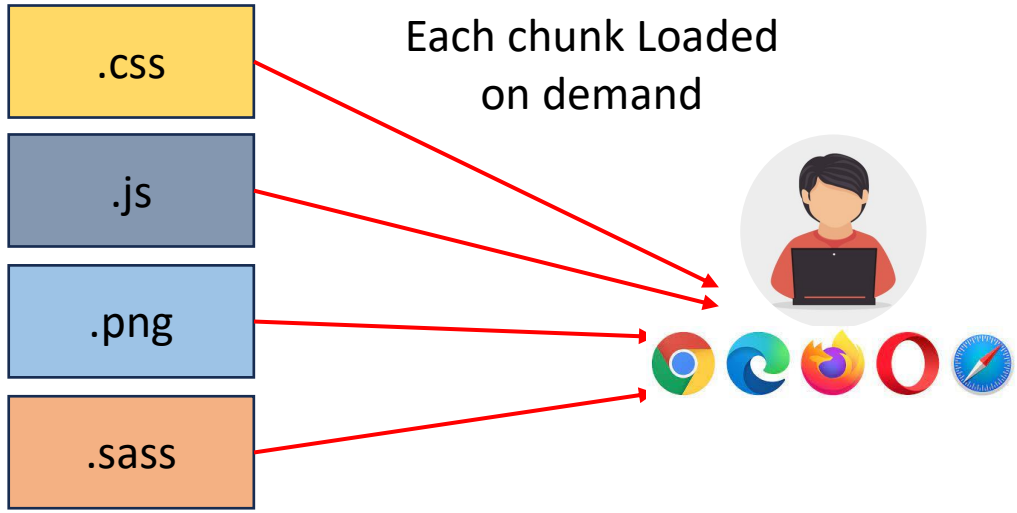
Q8. How do you inspect and analyze the generated chunks in a React application?

Q. What is Code Splitting in React? V. IMP.

- Code splitting is a technique to split the JavaScript bundle into **smaller chunks**, which are **loaded on-demand**.



Without Code Splitting



With Code Splitting Implemented

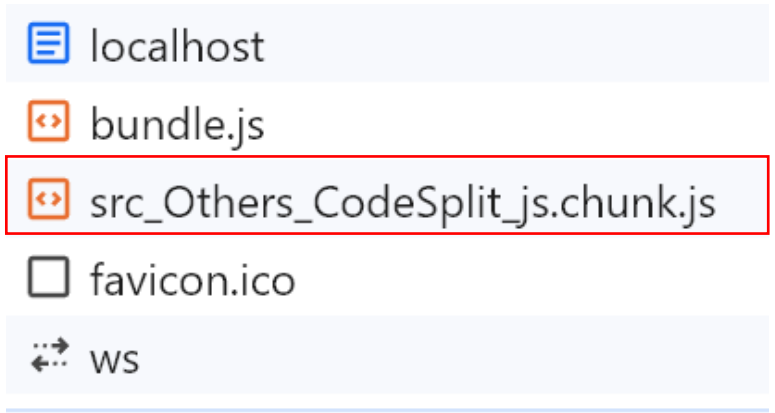
Q. How to Implement Code Splitting in React? V. IMP.

❖ **3 steps for Code splitting in React:**

1. Use **React.lazy()** to lazily import components.
2. Wrap components with **Suspense** to handle loading.
3. Configure your build tool (e.g., **Webpack**) for dynamic imports.

Q. How to Implement Code Splitting in React? V. IMP.

```
// CodeSplit.js: Component
const CodeSplit = () => {
  return <div>My component!</div>;
};

export default CodeSplit;
```




```
// App.js
import React, { lazy, Suspense } from "react";

const CodeSplit = lazy(() => import("./Others/CodeSplit"));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <CodeSplit />
      </Suspense>
    </div>
  );
}
export default App;
```

```
// webpack installation command
// npm install webpack webpack-cli --save-dev

// webpack.config.js: under root
const path = require('path');

module.exports = {
  // other webpack configuration...
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

Q. What is the role of Lazy and Suspense methods in React? V. IMP.

- React.lazy is a function that allows you to **load a component lazily**.
- It enables code splitting by allowing you to import a component asynchronously/ dynamically, meaning component is loaded only when needed only.

- The Suspense component is used to display a fallback UI while the lazily loaded component is being fetched.

```jsx
// App.js
import React, { lazy, Suspense } from "react";

const CodeSplit = lazy(() => import("./Others/CodeSplit"));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <CodeSplit />
      </Suspense>
    </div>
  );
}
export default App;
```

Q. What are the Pros and Cons of Code Splitting?

5 Pros of Code Splitting:

1. Faster Initial Load Time:	Code splitting reduces the initial load time of your application by only loading the necessary code for the current view or feature. Good for performance.
2. Optimized Bandwidth Usage:	By loading only, the code needed for a specific page, it reduces the amount of data transferred over the network. Good for slow network.
3. Improved Caching:	Smaller, more focused code chunks are more likely to be cached by the browser.
4. Parallel Loading:	Multiple smaller chunks can be loaded simultaneously, leading to faster overall loading times.
5. Easier Maintenance:	Code splitting can make your codebase more modular, independent and easier to maintain.

Q. What are the Pros and Cons of Code Splitting?

5 Cons of Code Splitting:

1. Complexity:	Implementing code splitting introduces additional complexity to your application. This complexity can make the development process slow.
2. Tooling Dependencies:	Proper code splitting often requires specific build tools and configurations, such as Webpack and Babel. Managing these tools is challenging.
3. Potential for Runtime Errors:	Dynamically loading code at runtime can introduce the possibility of runtime errors. Careful testing is necessary to catch such issues.
4. Increased Number of Requests:	Code splitting may increase the number of HTTP requests needed to fetch all the necessary chunks. This can impact performance.
5. Learning Curve:	Developers who are new to code splitting may need time to understand the concepts and best practices. This can be a challenging.

Q. What is the role of the import() function in code splitting?

- The import() function returns a promise that allow **dynamic loading of modules**.

```
// App.js
import React, { lazy, Suspense } from "react";

const CodeSplit = lazy(() => import("./Others/CodeSplit"));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <CodeSplit />
      </Suspense>
    </div>
  );
}
export default App;
```

Q. What is the purpose of the fallback prop in Suspense?

- The fallback prop provides a **loading indicator** or UI while the dynamically imported component is being loaded.

```jsx
// App.js
import React, { lazy, Suspense } from "react";

const CodeSplit = lazy(() => import("./Others/CodeSplit"));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <CodeSplit />
      </Suspense>
    </div>
  );
}
export default App;
```

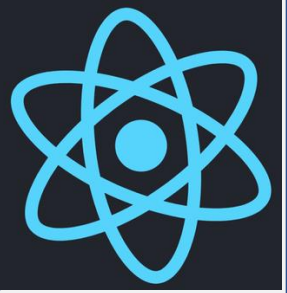
Q. Can you dynamically load CSS files using code splitting in React?

- **Yes**, using dynamic import() for CSS files allows you to load styles on-demand along with the corresponding components.

Q. How do you inspect and analyze the generated chunks in a React application?

- Use tools like **Webpack Bundle Analyzer** to analyze the size and composition of chunks.

15: React - Others



Q1. What is a Higher-Order Component in React? V. IMP.

Q2. What are the 5 ways to Style React components? Explain Inline Styles?

Q3. What are the differences between React & React Native?

Q4. What is GraphQL?

Q5. What are the Top 3 ways to achieve state management? When to use what in React?

Q6. How can you Implement Authentication in a React application? V. IMP.

Q7. What is the use of React Profiler?

Q8. What is the difference between fetch & axios for api calls in React?

Q9. What are the popular Testing Libraries for React?

Q10. How can you Optimize Performance in a React application? V. IMP.

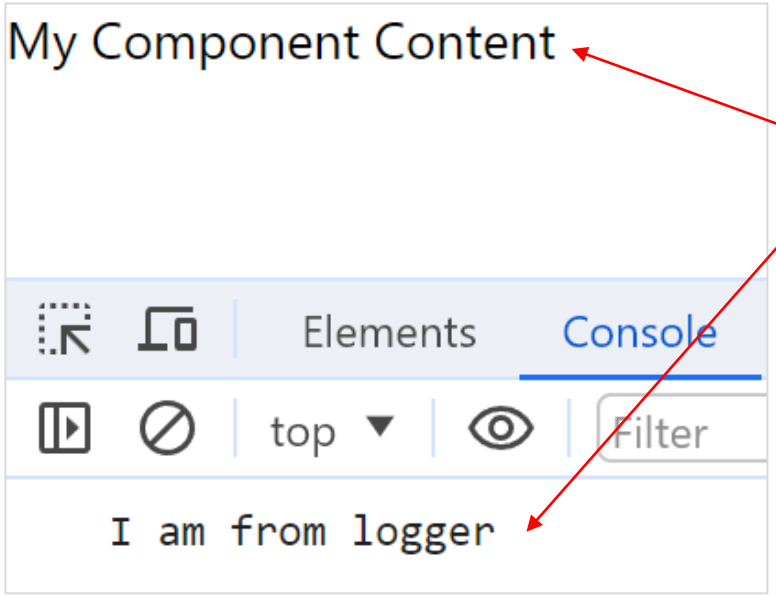
Q11. Explain Reactive Programming with example?

Q12. In how many ways can we implement Reactive Programming in React?

Q13. How to pass data from child component to parent Component in React?

Q. What is a Higher-Order Component in React? V. IMP.

- A Higher-Order Component is a component which takes **another component as an argument** and adds extra features to another component.
- HOC can be used for providing **logging functionality** to all the components in a reusable way.




```
// HocLogger.js: High Order Component
const HocLogger = (HocUse) => {

  return function WithLogger(props) {
    console.log("I am from logger");
    return <HocUse />;
  };

};
export default HocLogger;
```

```
// HocUse.js: Normal Component will HOC
const HocUse = () => {
  return <div>My Component Content</div>;
};

export default Hoc(HocUse);
```

```
// index.js
root.render(<HocUse></HocUse>);
```

Q. What are the 5 ways to Style React components? Explain Inline Styles?

5 ways to Style React components

1. Inline Styles
2. CSS Stylesheets
3. CSS-Modules
4. Global Stylesheets
5. CSS Frameworks

```jsx
import React from "react";

const AppInlineStyle = () => {
  return (
    <div>
      <h1 style={inlineStyles.title}>Inline</h1>
    </div>
  );
};

const inlineStyles = {
  title: {
    color: "blue",
    fontSize: "24px",
  },
};

export default AppInlineStyle;
```

Inline

Q. What are the differences between React & React Native?

React	React Native
1. React is a library	React Native is a framework.
2. React is used for building web interfaces.	React Native is used for building mobile applications.
3. Run on Web browsers.	Run on iOS and Android platforms.
4. HTML and CSS are used for UI.	Native UI components (e.g., View, Text) are used for UI.
5. Deployed as web applications.	Deployed through app stores(e.g., App Store, Google Play).

Q. What is GraphQL?

- GraphQL is a **query language for APIs** (Application Programming Interfaces) and a runtime for executing those queries with your existing data.

- GraphQL and React are often used together. React components can use GraphQL queries to fetch the data required for rendering.

GraphQL

```
<!--GraphQL query example -->
query {
  user(id: 1) {
    id
    name
    email
    posts {
      title
      content
    }
  }
}
```

Q. What are the Top 3 ways to achieve state management? When to use what in React?

1. useState Hook:

- **When to use:** Simple component-level state.
- **Reason:** Ideal for applications having small components and isolated state because it is Lightweight and built into React only.

2. Context API:

- **When to use:** Prop drilling avoidance for sharing global data.
- **Reason:** Simplifies data passing through the component tree, reducing the need for manual prop drilling.

3. Redux:

- **When to use:** Large-scale applications with complex state.
- **Reason:** Centralized store and actions provide a predictable state management pattern, aiding in debugging and scalability.

Q. How can you Implement Authentication in a React application? V. IMP.

Front-end/ Client-side

Middleware/ Server-side/ Backend

1. POST: {username, password}

2. Authenticate & create JWT **Token**

3. Return Response {JWT token}

4. Store JWT token at local storage

5. Request Data {JWT token: Header}

6. Validate token signature

7. Send Data

8. Display data on browser

Q. What is the use of React Profiler?

- React Profiler is a set of tools in React that allows developers to profile(analyze) the **performance** of a React application.

```
// Wrap the section of code you want to profile
// with the React.Profiler component.
<React.Profiler id="example" onRender={callback}>
  {/* Your code to profile */}
</React.Profiler>;

// Define a callback function (onRender) that
// will be called whenever the component tree
// within the Profiler is committed.
function callback(id,phase,actualDuration,baseDuration,
  startTime,commitTime
) {
  // Process profiling data
  // Start time, End Time, Execution Time
}
```

Q. What is the difference between fetch & axios for api calls in React?

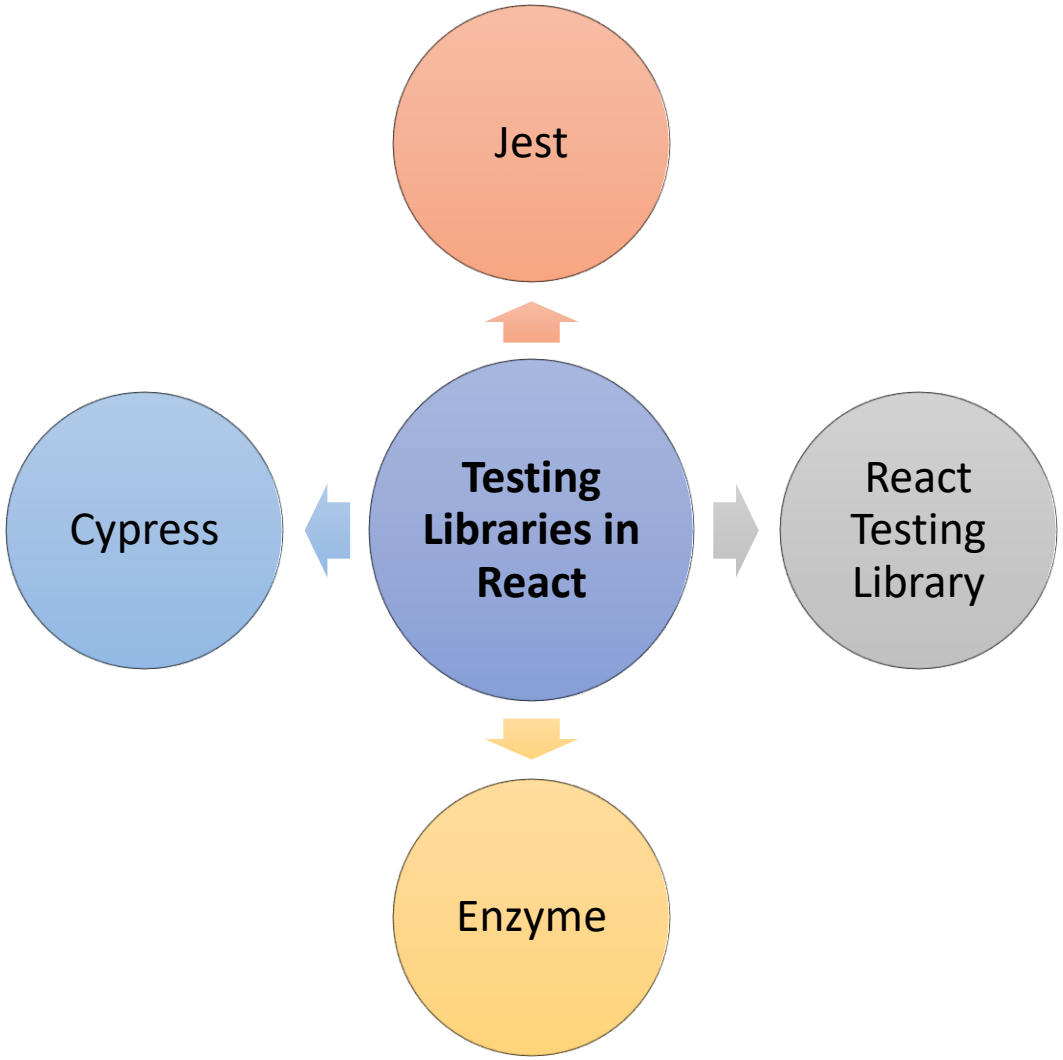
```
fetch("https://api.example.com/data")
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

1. fetch is a **built-in JavaScript function**, so it doesn't require any additional libraries.
2. fetch **returns Promises**, making it easy to work with asynchronous code using async/await syntax.
3. If you want to keep **http requests simple,** fetch is a good choice.

```
// installation command: npm install axios
import axios from "axios";
axios
  .get("https://api.example.com/data")
  .then((response) => console.log(response.data))
  .catch((error) => console.error("Error:", error));
```

1. Axios is a **third-party library,** that simplifies the process of making HTTP requests.
2. Axios allows you to **use interceptors**, which can be good for tasks like request/response logging, authentication, and error handling.
3. If you want to intercept http request/response, or improve error handling then Azios has more features to do it.

Q. What are the popular Testing Libraries for React?



Q. How can you Optimize Performance in a React application?

1. Memoization with useMemo and useCallback:
2. Optimizing Renders with React.Fragment:
3. Lazy Loading with React.lazy:
4. Code Splitting:
5. Optimizing Images and Assets

Q. How can you Optimize Performance in a React application?

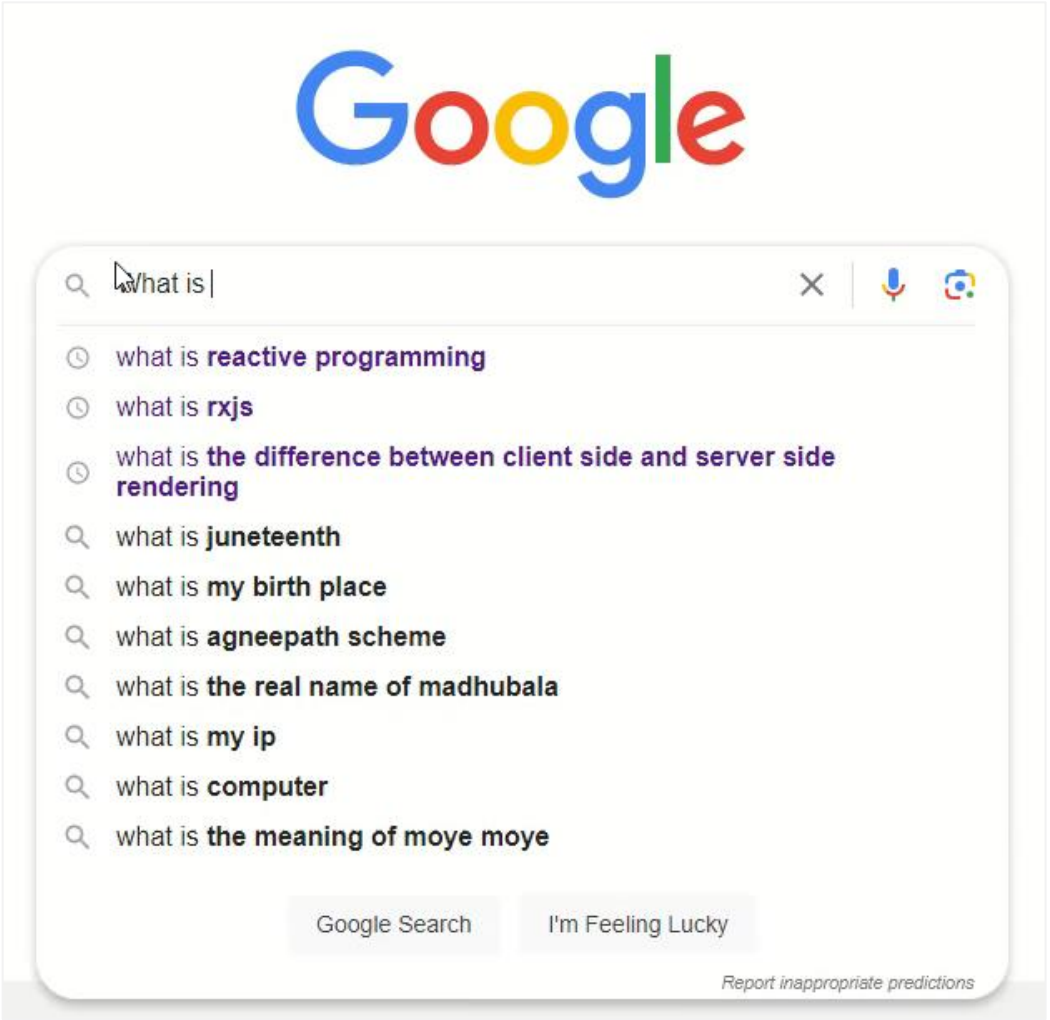
1. Memoization with useMemo and useCallback:	Use this hooks to memoize values and, reducing unnecessary recalculations.
2. Optimizing Renders with React.Fragment:	Use it to avoid unnecessary wrapper elements that could cause additional DOM nodes.
3. Lazy Loading with React.lazy:	Use it to load components lazily, reducing the initial bundle size and improving initial loading performance.
4. Code Splitting:	Employ code splitting to divide your application into smaller chunks that are loaded on demand, improving initial load times.
5. Optimizing Images and Assets:	Compress and optimize images, use responsive images, and leverage lazy loading for images to reduce network and rendering overhead.

Q. Explain Reactive Programming with example?

- Reactive programming is a programming paradigm that focuses on **reacting to changes and events** in a declarative and asynchronous manner.

- Declarative means a programming style where you write the code for what you want to achieve, rather than specifying step-by-step how to achieve it. For example, JSX in React has declarative syntax.

- Asynchronously means an action that does not block other actions.



Live Search Suggestion

Q. In how many ways can we implement Reactive Programming in React?

1. State and Props	Reacting to changes in local component state and passing data reactively through props.
2. React Hooks:	Leveraging **useState** and **useEffect** hooks for managing state and side effects in functional components.
3. Event Handling:	Reacting to user interactions through event handling and updating state accordingly.
4. Context API:	Sharing and managing global state reactively across components using the Context API.
5. Redux:	Using state management libraries like Redux for managing complex application state reactively.
6. Component Lifecycle Methods:	Using class components and lifecycle methods for handling side effects and updates.
7. Async/Await:	Utilizing **async/await** syntax for handling asynchronous operations reactively.
8. RxJS and Observables:	Leveraging RxJS for handling asynchronous operations and data streams in a reactive manner.

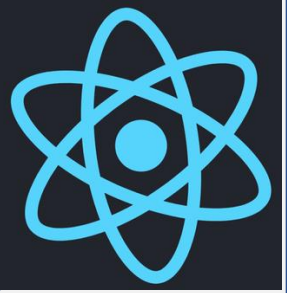
Q. How to pass data from child component to parent Component in React?

- Parent provides a **callback function** to child and then child component can then invoke this callback to pass data back to the parent.

```jsx
const ParentComponent = () => {
  // Callback to receive data from the child
  const callback = (data) => {
    console.log("Data from Child:", data);
  };

  return (
    <div>
      {/* Pass the callback to the child */}
      <ChildComponent fromChild={callback} />
    </div>
  );
};
export default ParentComponent;
```

```jsx
const ChildComponent = ({ fromChild }) => {
  // No local state in the child component
  const dataToParent = () => {
    // Directly pass the input value to the parent
    fromChild(document.getElementById
      ("inputField").value);
  };

  return (
    <div>
      <input type="text" id="inputField" />
      <button onClick={dataToParent}>Send</button>
    </div>
  );
};
export default ChildComponent;
```

16: Redux - Component/ Action/ Store/ Reducer



Q1. What is the role of Redux in React? **V. IMP.**

Q2. When to use Hooks and when to use Redux in React applications?

Q3. What is the Flow of data in React while using Redux? **V. IMP.**

Q4. How to install Redux for React application?

Q5. What are Action Creators in React Redux? **V. IMP.**

Q6. Difference between Action Creators, Action Object & Action Type?

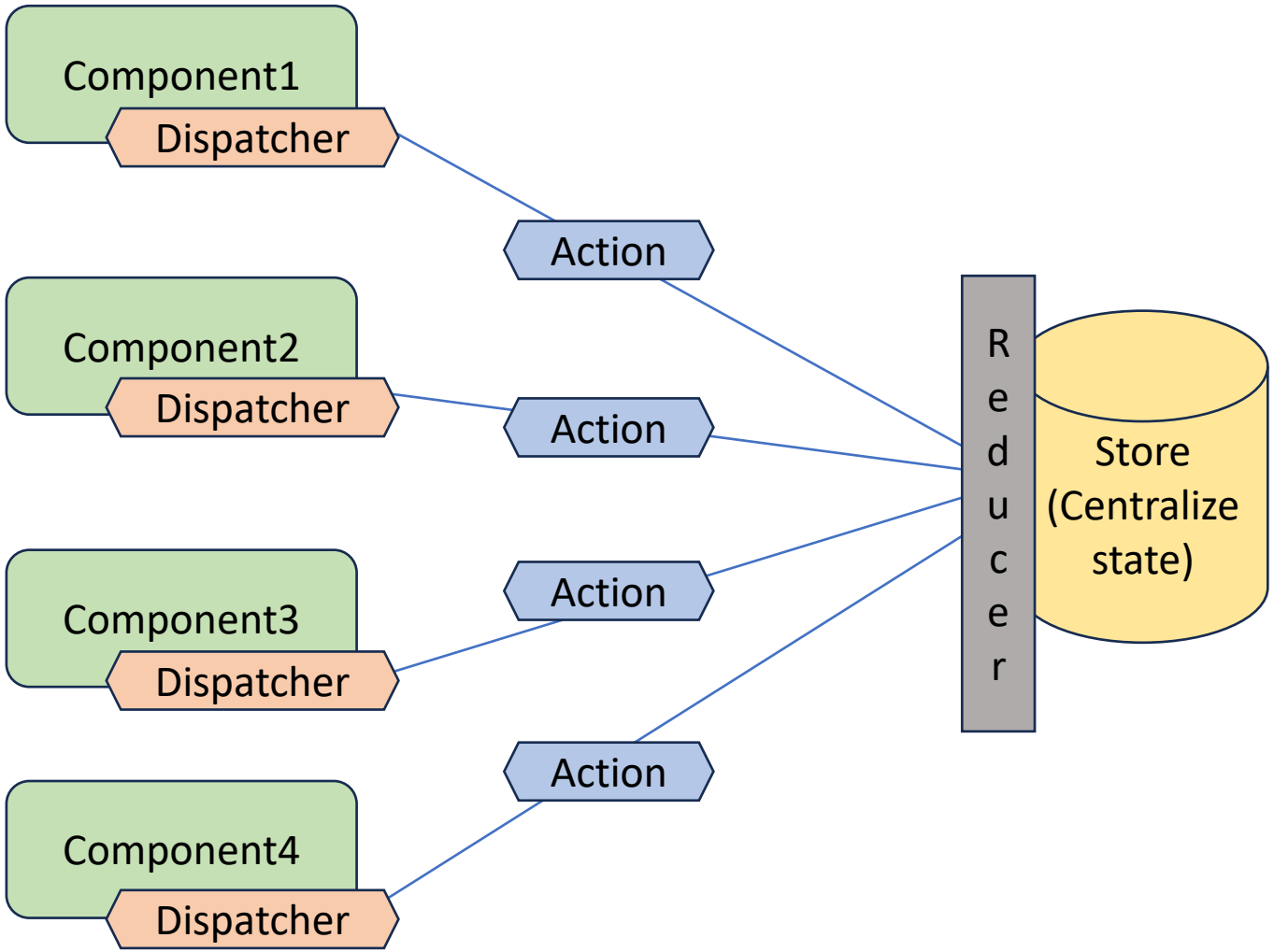
Q7. Explain React Component Structure while using Redux?

Q8. What is the role of Store in React Redux? **V. IMP.**

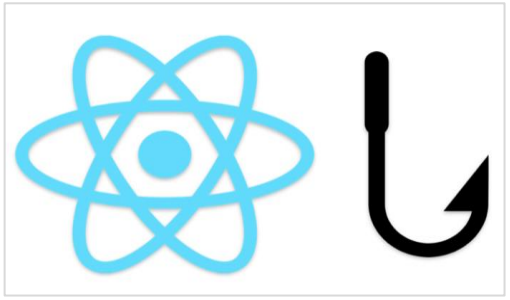
Q9. What is the role of Reducer in Redux? **V. IMP.**

Q. What is the role of Redux in React? V. IMP.

- Redux is an open-source JavaScript library used for **state management**.
- Redux provides a **centralized store** that holds the entire state of an application and allows components to access and update the state in a predictable manner.

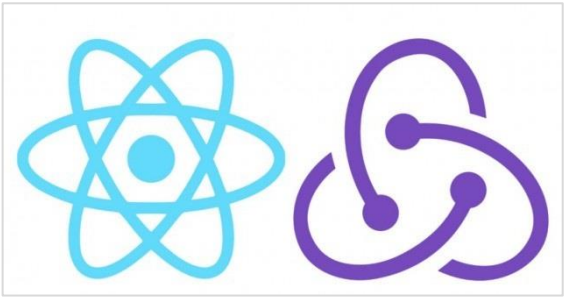


Q. When to use Hooks and when to use Redux in React applications? V. IMP.



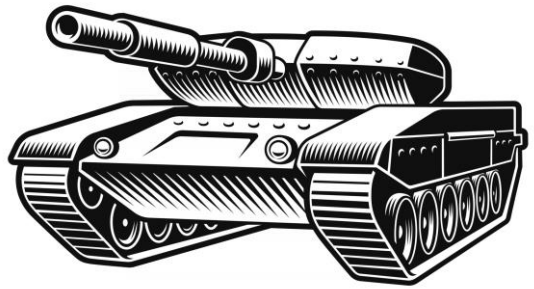
❖ **Use React with Hooks when:**

1. Your application is small or medium (5-50 Components)
2. Your application has simple state management.
3. State is specific to components.



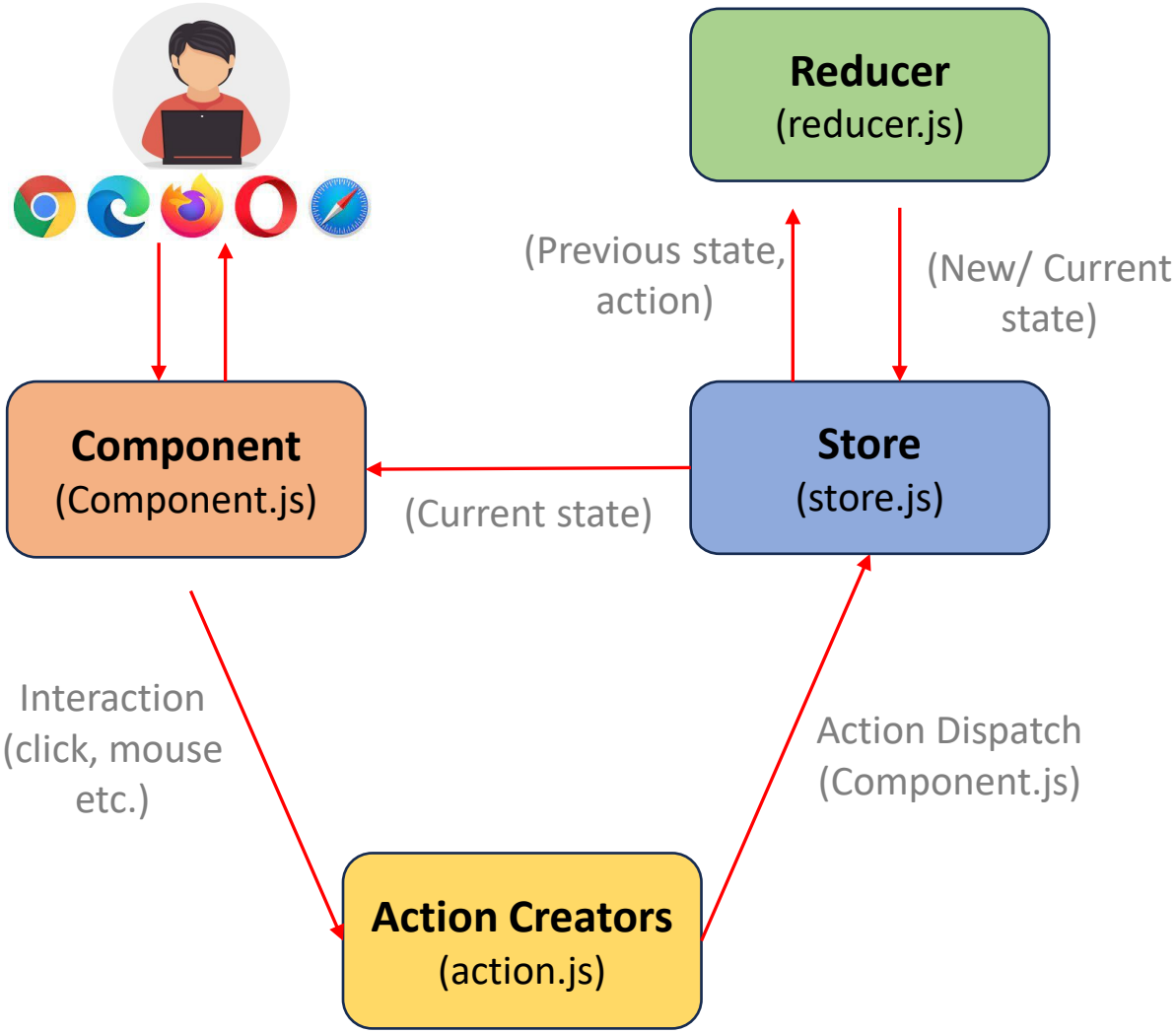
❖ **Use React with Redux when:**

1. Your application is big or complex (>50 Components)
2. Your application has complex global state management.
3. Share state among multiple components.



Q. What is the Flow of data in React while using Redux? V. IMP.

❖ **Flow of Data in React-Redux application**



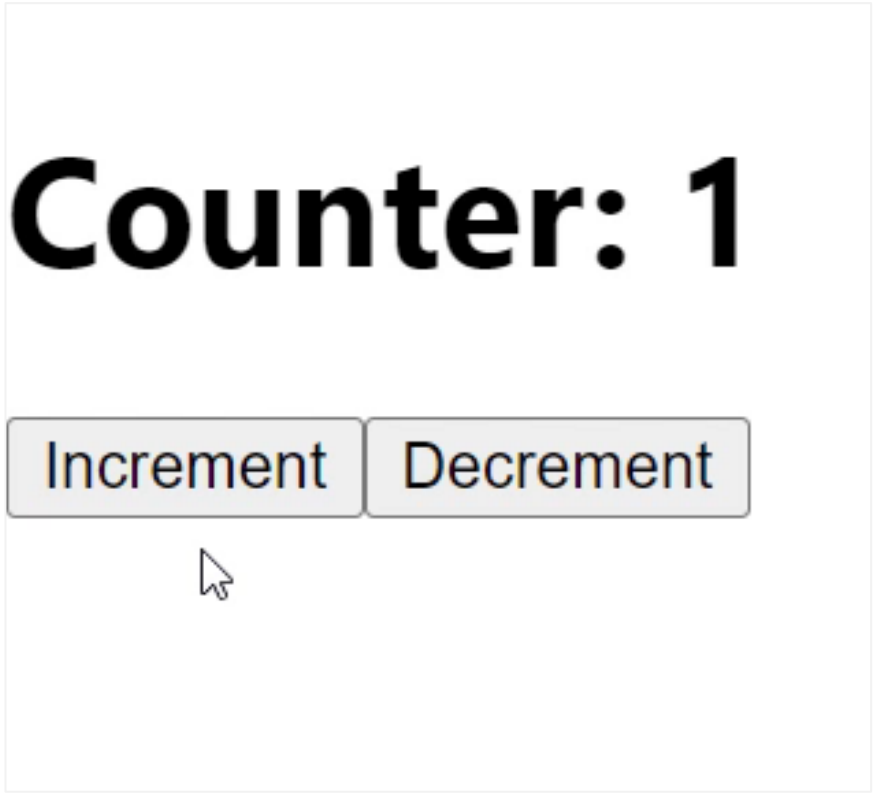
Q. How to install Redux for React application?

- Install the necessary packages using npm (run this command):

```
npm install redux react-redux
```

Q. What are Action Creators in React Redux? V. IMP.

- Action creators are functions that **create and return action objects**.




```
// actions.js

// Action creator function
export const increment = () => ({
  type: 'INCREMENT' // Action type
});

// Action creator function
export const decrement = () => ({
  type: 'DECREMENT' // Action type
});
```

Q. Difference between Action Creators, Action Object & Action Type? V. IMP.

- Action creator functions are functions that create and return action objects.

```
// Action creator
const increment = (amount) => {
  return {
    type: "INCREMENT",
    payload: {
      amount: amount,
    },
  };
};
```

- Action object are plain JavaScript objects returned by action creator functions.

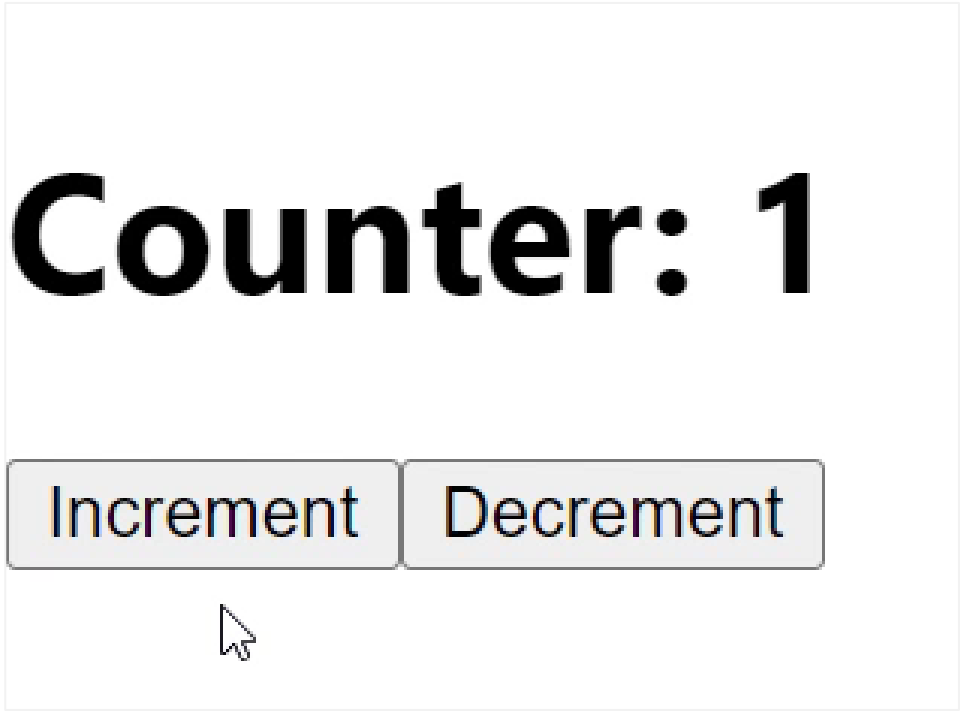
```
// Action object
const incrementAction = increment(2);
incrementAction = {
  type: "INCREMENT",
  payload: { amount: 2 },
};
```

- An action type is a string constant that defines the type of action.

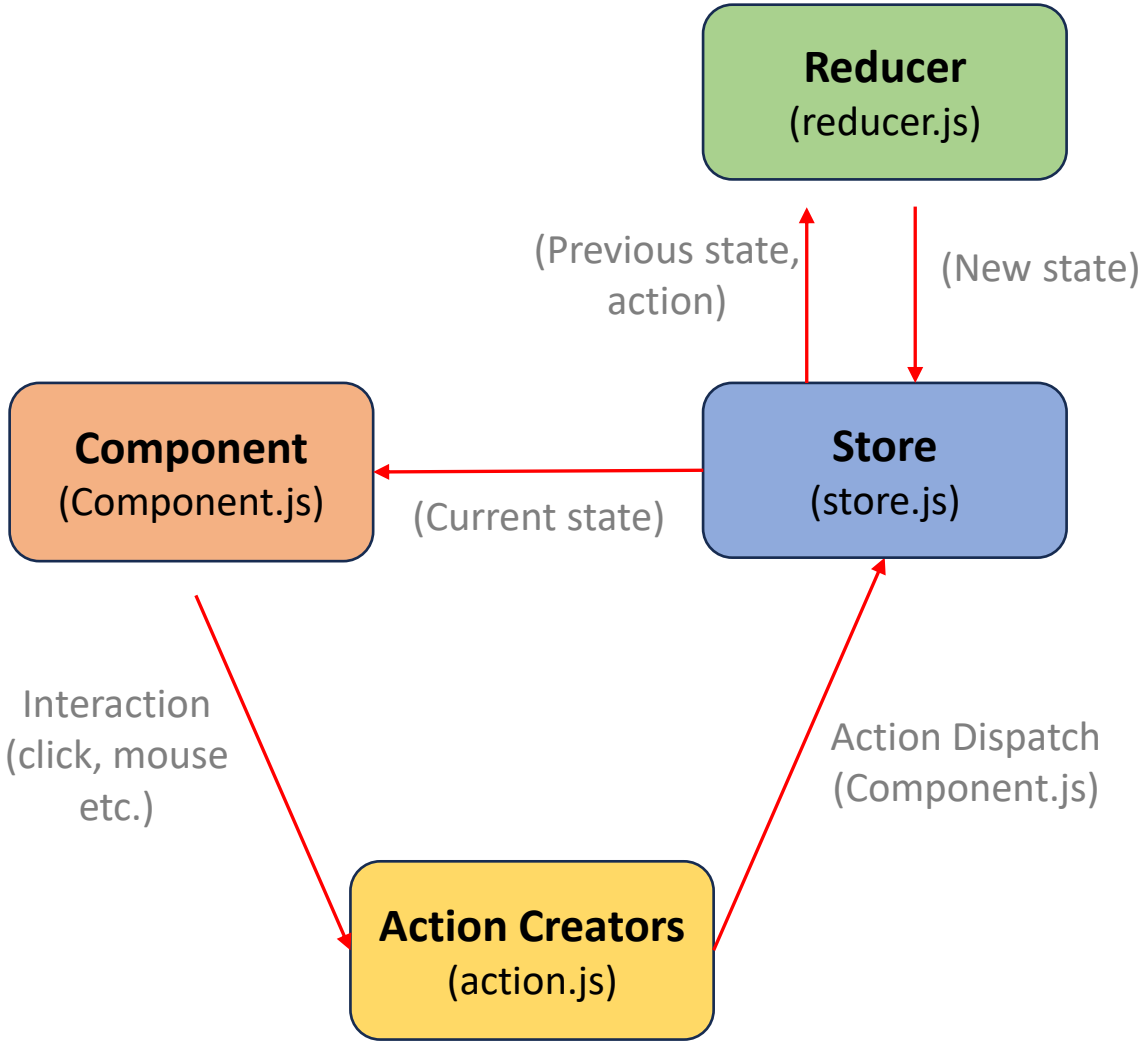
```
// Action type
var actionType = incrementAction.type;
```

Q. Explain React Component Structure while using Redux?

❖ **Example video of expected state management:**



❖ **Flow of Data in React-Redux application**



Q. Explain React Component Structure while using Redux?

```jsx
// CounterComponent.js

// 1. Import Statements
import React from "react";
import { connect } from "react-redux";
import { increment, decrement } from "./actions";

// 2. Functional Component
const CounterComponent = ({ count, increment, decrement }) => {
  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

```jsx
// 3. Map Redux actions to component props
const mapDispatchToProps = {
  increment,
  decrement,
};

// 4. Map Redux state to component props
const mapStateToProps = (state) => {
  return {
    count: state.counter.count, //From store
  };
};

// 5. Connect the component to Redux store
export default connect(mapStateToProps,
  mapDispatchToProps)(CounterComponent);
```

Q. Explain React Component Structure while using Redux?

❖ **5 Steps to implement Redux integration in component:**

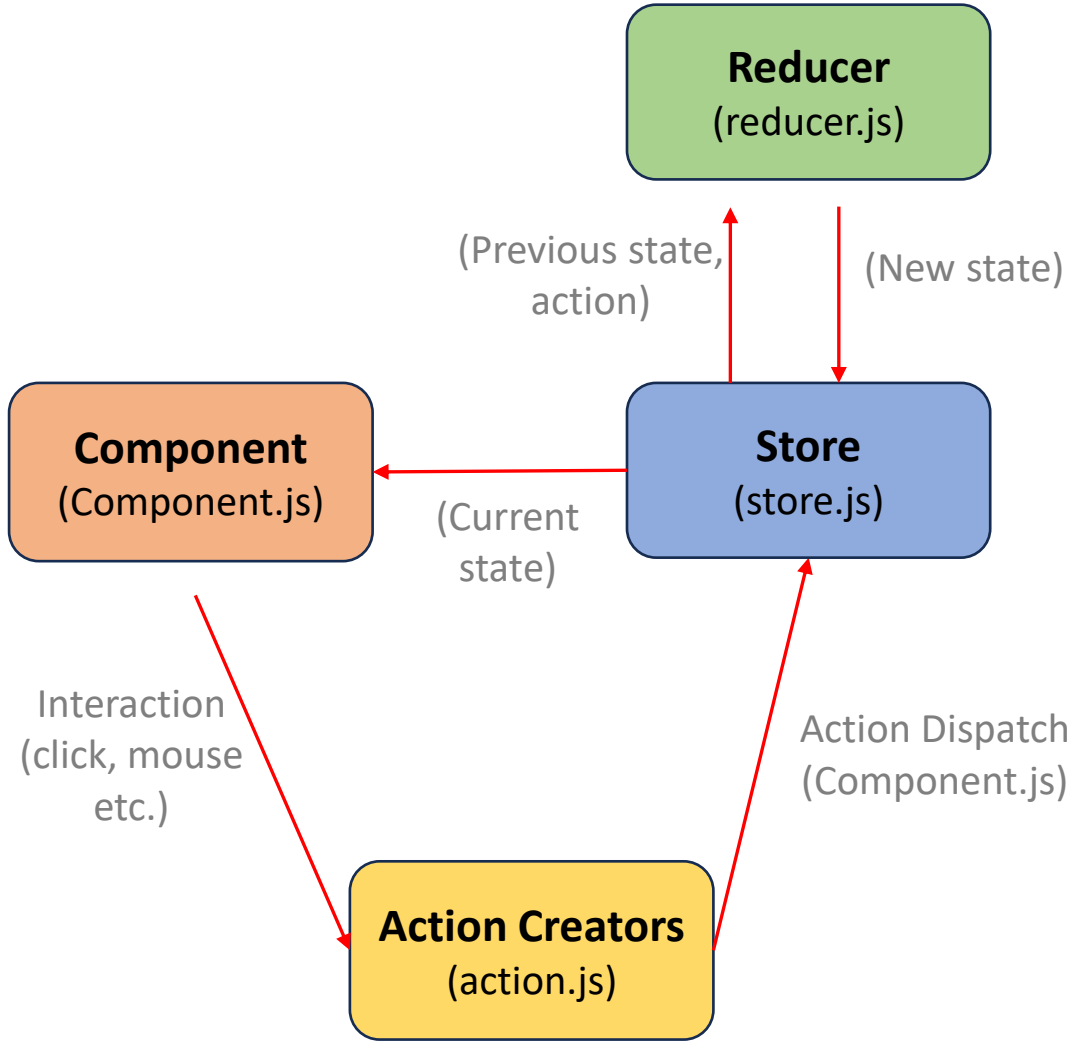
1. Import dependencies like connect function and action creators.
2. Define Functional Component to render & display elements.
3. Define function to dispatch action to reducer via store.
4. Define function to get state from Redux store back to component.
5. Connect both the above functions and the component to Redux store by connect() function.

Q. What is the role of Store in React Redux? V. IMP.

- Redux store enables the application to **update state using the defined reducer**.
- Redux Store is a **centralized place** for holding the state of all the components in the application.

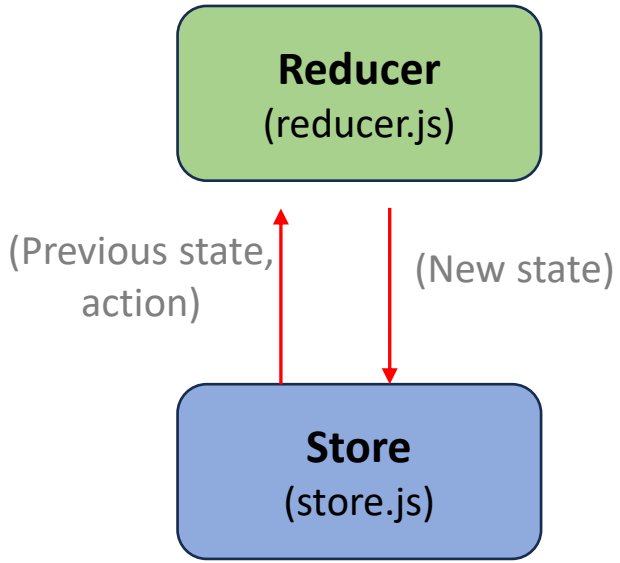
```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './reducer';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export default store;
```

- **Flow of Data in React-Redux application**



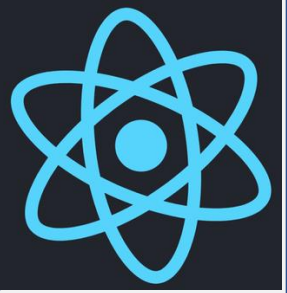
Q. What is the role of Reducer in Redux? V. IMP.

- A reducer is a function that takes the **previous state** and **an action** as arguments and returns the new state of the application.




```
// reducer.js
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};
export default counterReducer;
```

17: Redux - Core Principles/ Pros-Cons/ Local & Redux State



Q1. Explain the Core Principles of Redux? **V. IMP.**

Q2. List 5 benefits of using Redux in React? **V. IMP.**

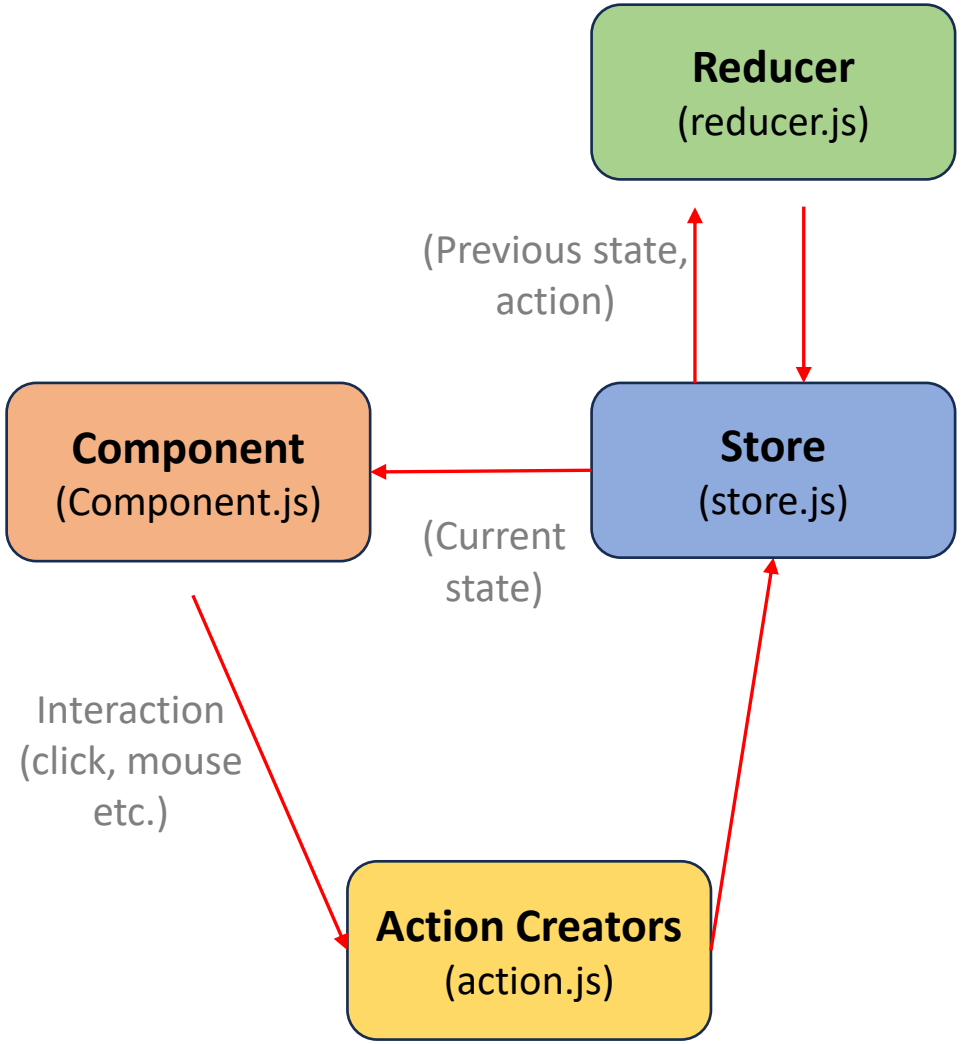
Q3. What are the differences between local component state & Redux state?

Q4. What are the challenges or disadvantages while using Redux?

Q. Explain the Core Principles of Redux? V. IMP.

1. Single Source of Truth (Store)
2. State is Read-Only (Unidirectional)
3. Changes using Pure Functions (Reducers)
4. Actions Trigger State Changes (Actions)
5. Predictable State Changes (Actions)

❖ **Flow of Data in React-Redux application**



Q. Explain the Core Principles of Redux? V. IMP.

1. Single Source of Truth:	The entire application state is stored in one place, simplifying data management and ensuring a consistent view of the application.
2. State is Read-Only:	State cannot be directly modified. To make changes to the state, you need to dispatch an action. This ensures that the state transitions are explicit and traceable.
3. Changes using Pure Functions (Reducers):	This ensures predictability and consistency because pure functions returns the same result if the same arguments are passed.
4. Actions Trigger State Changes:	Plain JavaScript objects (actions) describe state changes, guiding the store to invoke reducers and update the application state accordingly.
5. Predictable State Changes with Actions:	State changes are determined by actions, fostering a predictable flow of data and simplifying debugging in response to specific actions.

Q. List 5 benefits of using Redux in React? V. IMP.

1. Predictability & Centralization:	The application state is stored in a single, predictable and centralize source (the store).
2. Maintainability:	For larger application, Redux is a structured and scalable approach for managing state.
3. Debuggability:	Redux has powerful developer tools that make it easier to trace and debug state changes.
4. Interoperability:	Redux can be used with various JavaScript frameworks and libraries.
5. Community & Ecosystem:	Redux has a large and active community, resulting in a rich ecosystem of tools and extensions.

Q. What are the differences between local component state & Redux state?

Local Component State	Redux State
1. Scope: Limited to the component where defined.	Global and accessible across components.
2. Management: Managed internally by the component.	Managed externally by the Redux store.
3. Performance: Generally, more performant for small-scale applications.	More performant for large applications.
4. Complexity: Simpler to set up and manage.	Comparatively complex to manage.
5. Testing: Simpler to test with component-specific state.	Requires more comprehensive testing due to global nature and interactions between components.

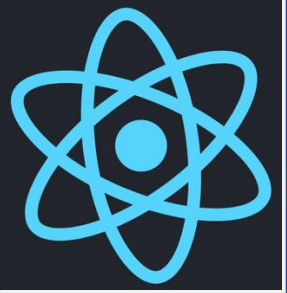
Q. What are the challenges or disadvantages while using Redux?

1. Boilerplate Code
2. Learning Curve
3. Verbosity and Complexity
4. Overhead for Small Projects
5. Global State for Local Components
6. Integration with Non-React Libraries

Q. What are the challenges or disadvantages while using Redux?

1. Boilerplate Code:	Implementing Redux requires writing extensive boilerplate code in action, reducer, store, increasing code volume and complexity.
2. Learning Curve:	Understanding Redux concepts can be challenging, posing a learning curve for developers, especially those new to React state management.
3. Verbosity and Complexity:	As projects grow, Redux code may become verbose and complex, demanding careful management of actions and reducers.
4. Overhead for Small Projects:	In small projects, Redux may introduce unnecessary complexity, potentially outweighing its benefits in development efficiency.
5. Global State for Local Components:	Overusing Redux for local state introduces unnecessary complexity, as not all state requires a global scope.
6. Integration with Non-React Libraries:	Integrating Redux with non-React libraries or frameworks may demand additional effort and customization, potentially adding complexity to the project.

18: Redux - Short Answer



Q1. What is Provider Component? How components getting the state from Redux store? V. IMP.

Q2. What is the role of Connect function in React-Redux? V. IMP.

Q3. What are the 4 Important Files in React-Redux project?

Q4. How to structure the project and maintain state in multiple components?

Q5. Explain the concept of immutability in the context of Redux?

Q6. Which are the typical properties of an Action object in React-Redux project?

Q7. Difference btw mapDispatchToProps & mapStateToProps in the connect?

Q8. What is the meaning of Unidirectional Data Flow in Redux?

Q9. How does Redux handle communication between components?

Q10. What is Payload property in Redux?

Q. What is Provider Component? How components getting the state from Redux store?

- Provider component of react-redux will **make the Redux store available** to all connected components.

```
// index.js
import { Provider } from "react-redux";
import store from "./Redux/store";

const root = ReactDOM.createRoot(
    document.getElementById("root"));
root.render(
  <Provider store={store}>
    <CounterComponent />
  </Provider>
);
```

```
// CounterComponent.js
// Map Redux state to component props
const mapStateToProps = (state) => {
  return {
    count: state.counter.count,
  };
};
```

Q. What is the role of Connect function in React-Redux?

- The connect function is used to **make the connection** between a React component and the Redux store.

```
import React from "react";
import { connect } from "react-redux";

// 3. Map Redux actions to component props
const mapDispatchToProps = {
  increment,
  decrement,
};

// 4. Map Redux state to component props
const mapStateToProps = (state) => {
  return {
    count: state.counter.count, //From store
  };
};

// 5. Connect the component to Redux store
export default connect(mapStateToProps,
  mapDispatchToProps)(CounterComponent);
```

Q. What are the 4 Important Files in React-Redux project?

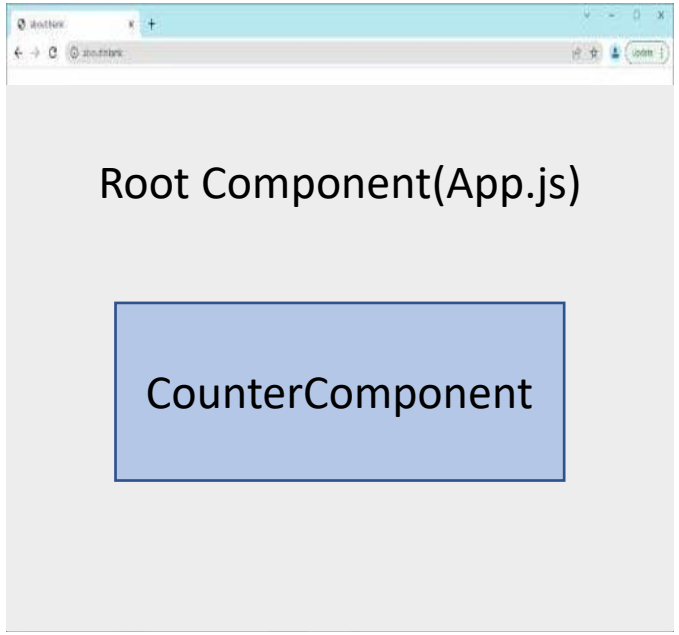
❖ Important File for Redux Implementation:

1. **actions.js:** Defines action creators.
2. **reducer.js:** Update the state based on the action and previous state received from the store.
3. **store.js:** Stores the updated state received from reducer and export it to the components.
4. **CounterComponent.js:** React component connected to Redux, uses state and actions to render UI and handle user interactions.

```
src/
|-- actions.js
|-- reducer.js
|-- store.js
|-- CounterComponent.js
|-- App.js
|-- index.js
```

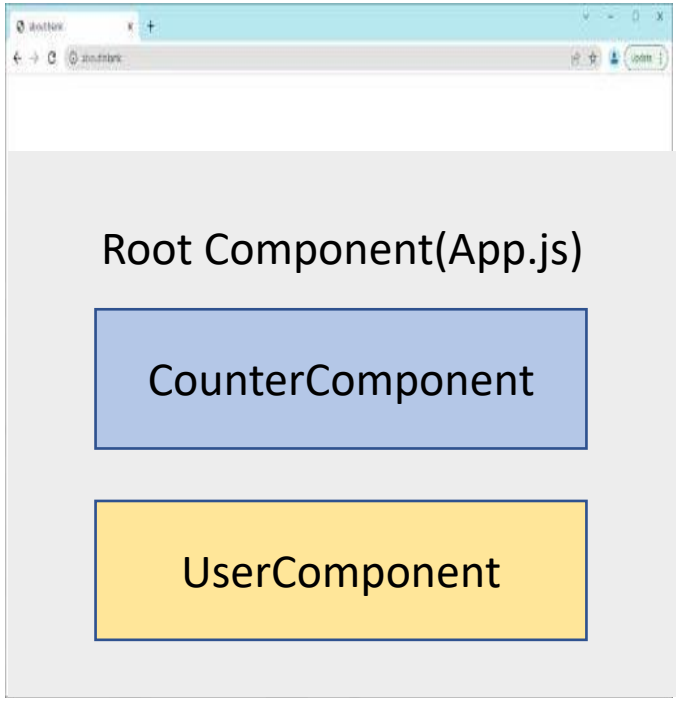
Q. How to structure the project and maintain state in multiple components?

- Single component project structure




```
src/
 |-- actions.js
 |-- reducer.js
 |-- store.js
 |-- CounterComponent.js
 |-- App.js
 |-- index.js
```

- Multiple component project structure




```
src/
 |-- actions/
 |    |-- counterActions.js
 |    |-- userActions.js
 |
 |-- reducers/
 |    |-- counterReducer.js
 |    |-- userReducer.js
 |
 |-- store/
 |    |-- configureStore.js
 |
 |-- components/
 |    |-- CounterComponent.js
 |    |-- UserComponent.js
 |
 |-- App.js
 |-- index.js
```

Q. Explain the concept of immutability in the context of Redux?

- In Redux, immutability is the principle that once an object (such as the state which is readonly) is created, it **cannot be changed** or modified directly.
- To make changes to the state, you need to dispatch an action.

Q. Which are the typical properties of an Action object in React-Redux project?

- 2 Typical properties in most of the React-Redux project:

1. **Type:** Type describes the type of action being performed.
2. **Payload:** Carries data from mostly from external API's.

```
// Action creator
const increment = (amount) => {
  return {
    type: "INCREMENT",
    payload: {
      amount: amount,
    },
  };
};
```

```
// Action object
const incrementAction = increment(2);
// incrementAction = {
//  type: "INCREMENT",
//  payload: { amount: 2 }
// }
```

Q. Difference btw mapDispatchToProps & mapStateToProps in the connect?

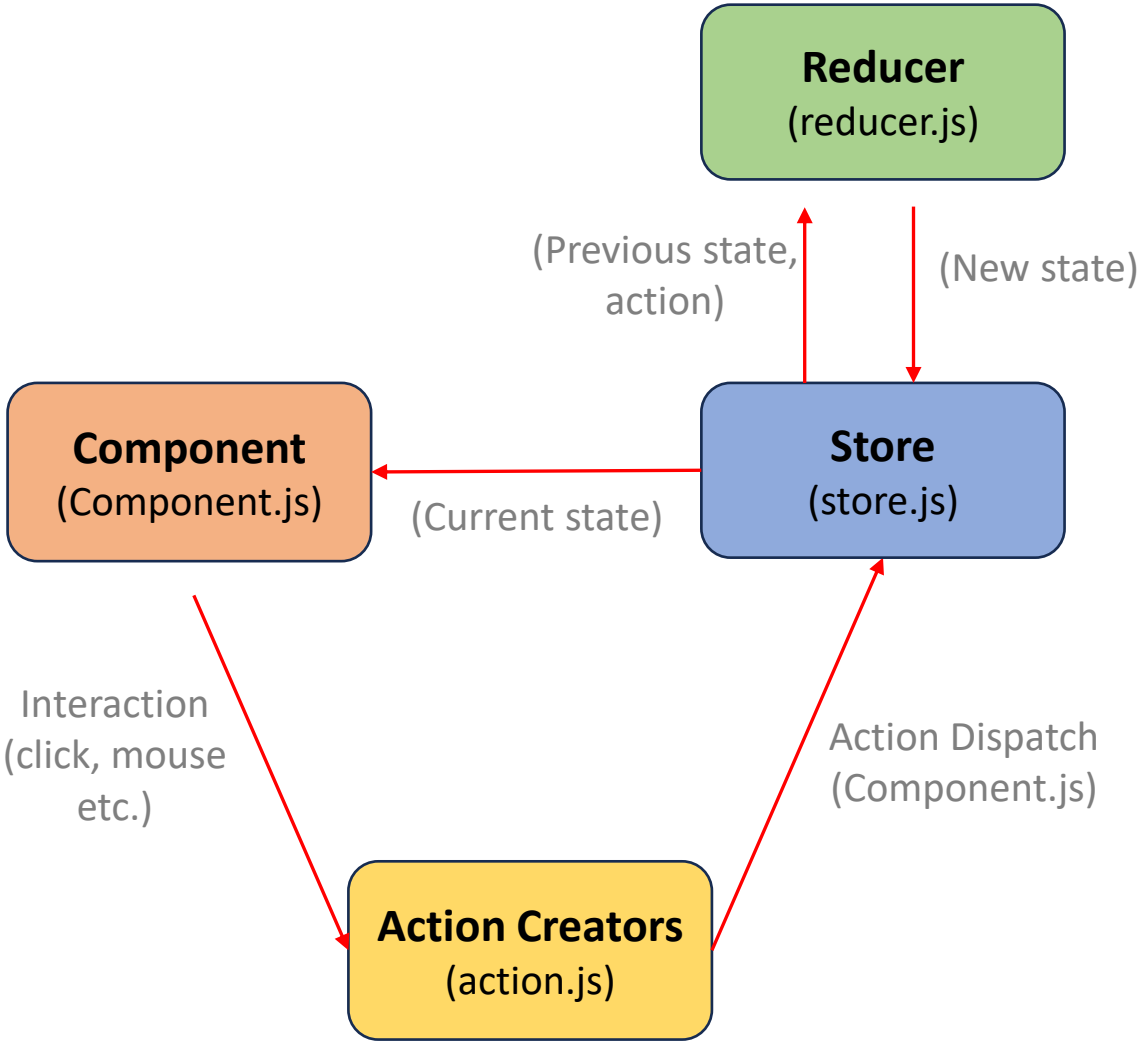
- mapDispatchToProps() function dispatches action to the store to **update the state** using a reducer.
- mapStateToProps() function **gets the state** for the component from Redux store.

```
// 3. Map Redux actions to component props
const mapDispatchToProps = {
  increment,
  decrement,
};

// 4. Map Redux state to component props
const mapStateToProps = (state) => {
  return {
    count: state.counter.count, //From store
  };
};

// 5. Connect the component to Redux store
export default connect(mapStateToProps,
  mapDispatchToProps)(CounterComponent);
```

Q. What is the meaning of Unidirectional Data Flow in Redux?

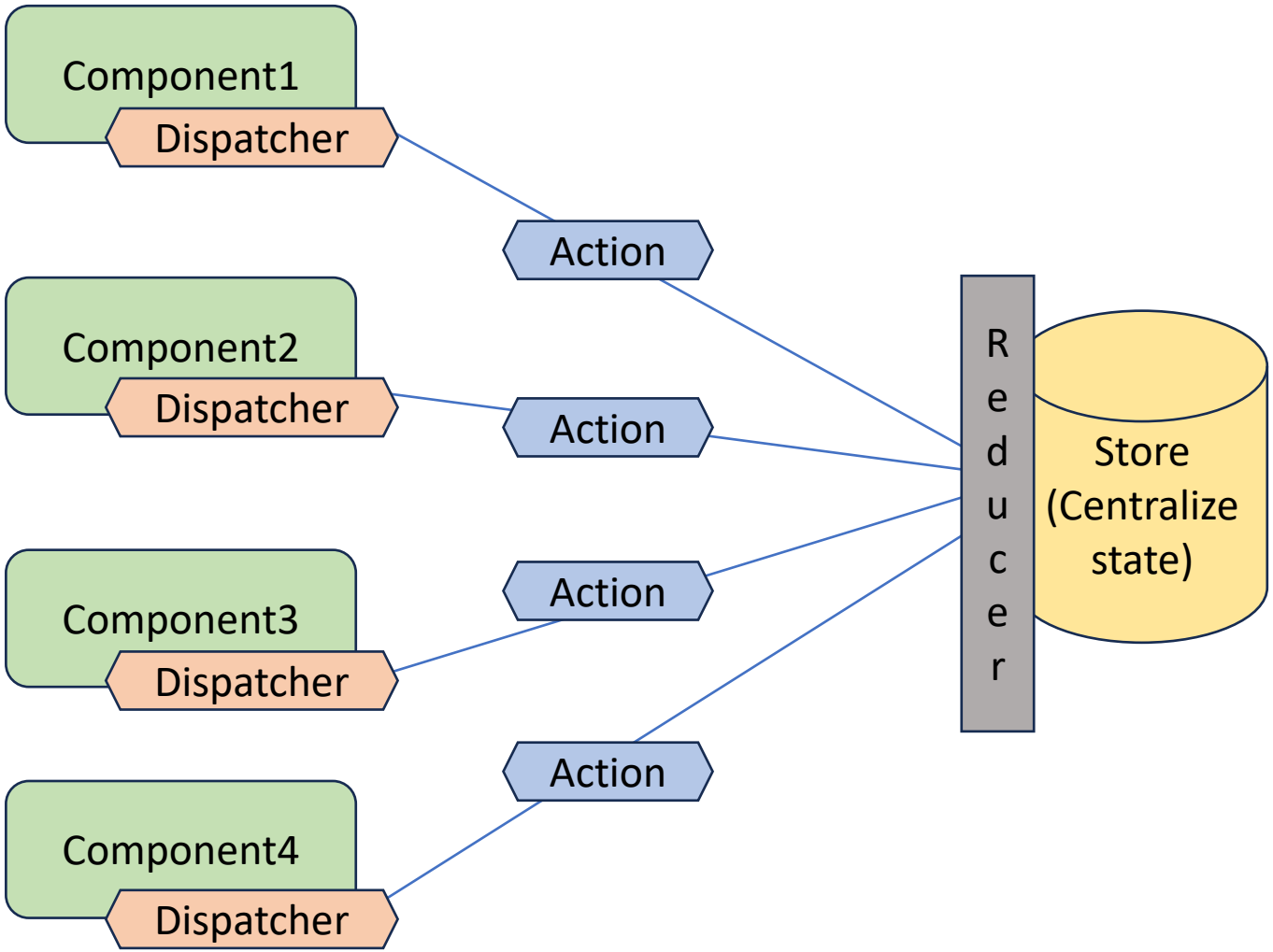
- In a unidirectional data flow architecture, data follows a single, clear path.
- Redux enforces this pattern to provide a predictable state management mechanism.

- **Flow of Data in React-Redux application**



Q. How does Redux handle communication between components?

- Redux handles communication between components through a **centralized state management system(store)**.

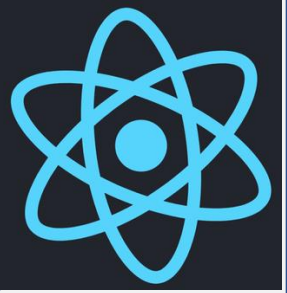


Q. What is Payload property in Redux?

- Along with the action type, payload property in Redux actions **holds the data** being sent to the reducer.
- Payload contains information necessary for the reducer to update the state based on the action.

```
// Action creator
const increment = (amount) => {
  return {
    type: "INCREMENT",
    payload: {
      amount: amount,
    },
  };
};
```

19: Redux - Thunk/ Middleware/ Error Handling/ Flux



Q1. What is the difference between Regular Action creator & Thunk action creator?

Q2. Explain the concept of Middleware in React-Redux?

Q3. How can you handle Asynchronous Operations & side-effect in React-Redux?

Q4. How does Error Handling work in Redux?

Q5. What is the difference between Flux & Redux?

Q. What is the difference between Regular Action creator & Thunk action creator?

- A regular action creator **returns a plain JavaScript object** with a type property, describing the type of action to be performed.

```
// Regular action creator
export const increment = () => ({
  type: "INCREMENT",
});
```

- 3 main points about Thunks:

1. A Thunk action creator is a function that **returns another function**.
2. This returned function dispatched to the Redux store.
3. Thunks are used to handle complex asynchronous operations, such as data fetching or API calls, based on which the action will be decide.

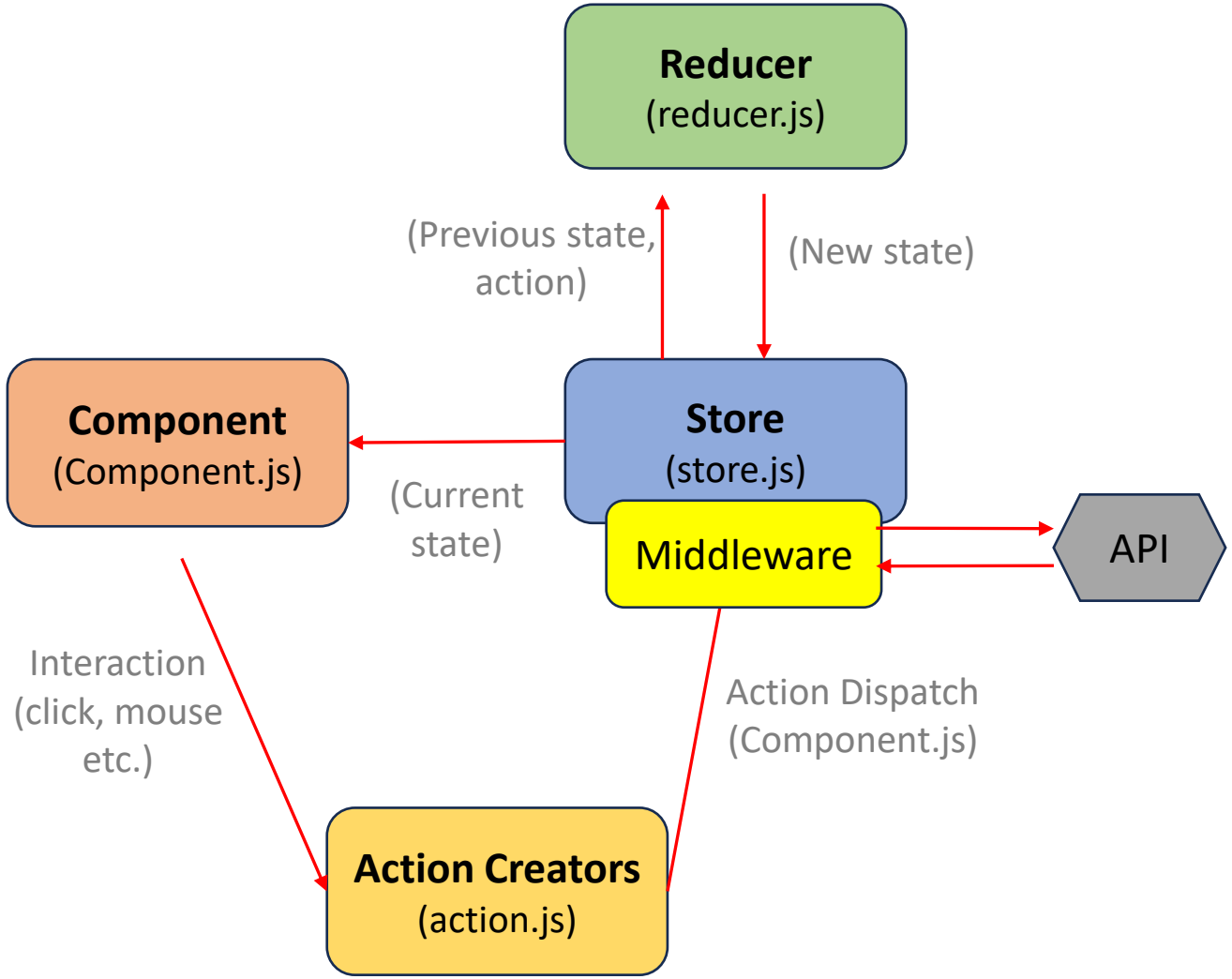
```
// Thunk action creator
export const fetchData = () => (dispatch) => {
  // Asynchronous logic, e.g., API call using fetch
  fetch("https://api.example.com/data")
    .then((response) => response.json())
    .then((data) => dispatch({ type: "SUCCESS", payload: data }))
    .catch((error) =>
      dispatch({ type: "FAILURE", payload: error.message })
    );
};
```

Q. Explain the concept of Middleware in React-Redux?

- Middleware provides a mechanism to **add extra functionality** to the Redux store.
- Middleware can intercept actions, modify them, or execute additional logic in actions before they reach the reducers.

```
// Configure the Redux store
// with middleware
const store = configureStore({
  reducer: rootReducer,
  middleware: [thunk],
});
```

- Flow of Data in React-Redux application with Middleware



Q. How can you handle Asynchronous Operations & side-effect in React-Redux?

- Use middleware and Redux Thunk action creators **to handle asynchronous operations**.

```
// Configure the Redux store
// with middleware
const store = configureStore({
  reducer: rootReducer,
  middleware: [thunk],
});
```

```
// Thunk action creator
export const fetchData = () => (dispatch) => {
  // Asynchronous logic, e.g., API call using fetch
  fetch("https://api.example.com/data")
    .then((response) => response.json())
    .then((data) => dispatch({ type: "SUCCESS", payload: data }))
    .catch((error) =>
      dispatch({ type: "FAILURE", payload: error.message })
    );
};

// Usage in a component
dispatch(fetchData());
```

Q. How does Error Handling work in Redux?

- Error handling in Redux can be done by using **try-catch blocks** in action creator, middleware and reducers.

```
export const fetchData = () => async (dispatch) => {
  try {
    const data = await api.fetchData();
    dispatch({ type: "FETCH_SUCCESS", payload: data });
  } catch (error) {
    dispatch({ type: "FETCH_ERROR", payload: error.message });
  }
};
```

Q. What is the difference between Flux & Redux?

- Flux is an **architecture pattern**. It provides a set of design principles for organizing code to manage state in a unidirectional data flow.
- Redux is a library that implements the Flux architecture.

20: JavaScript Essentials for React

Q1. What are Variables? What is the difference between var, let, and const ?

Q2. What are the Types of Conditions statements in JS?

Q3. What is Error Handling in JS?

Q4. What is the difference between Spread and Rest operator in JS?

Q5. What are Arrays in JS? How to get, add & remove elements from arrays?

Q6. What is Array Destructuring in JS?

Q7. What are Functions in JS? What are the types of function?

Q8. Difference between Named and Anonymous functions? When to use what?

Q9. What is Function Expression in JS?

Q10. What are Arrow Functions in JS? What is it use?

Q11. What are Callback Functions? What is it use?

Q12. What is Higher-order function In JS?

Q13. What are Pure and Impure functions in JS?

20: JavaScript Essentials for React

Q14. What are template literals and string interpolation in strings?

Q15. What are Objects in JS?

Q16. What is the difference between an array and an object?

Q17. How do you add or modify or delete properties of an object?

Q18. What is asynchronous programming in JS? What is its use?

Q19. What is the difference between synchronous and asynchronous programming?

Q20. What are Promises in JavaScript?

Q21. How to implement Promises in JavaScript?

Q22. What is the purpose of async/ await? Compare it with Promises?

Q23. What are Classes in JS?

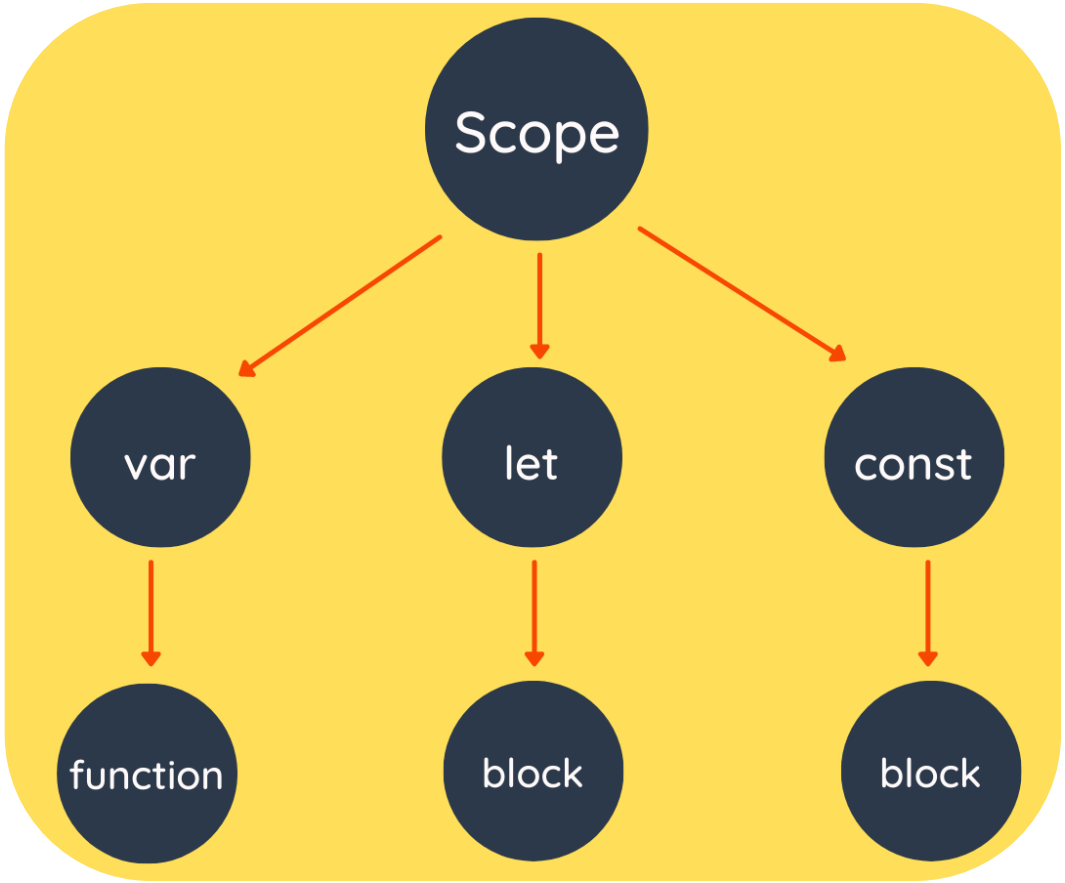
Q24. What is a Constructor?

Q25. What is the use of this keyword?

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- Variables are used to **store** data.

```
var count = 10;
```



Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- var creates a **function-scoped** variable.

```
//using var
function example() {

  if (true) {

    var count = 10;
    console.log(count);
    //output: 10
  }

  console.log(count);
  //Output: 10
}
```

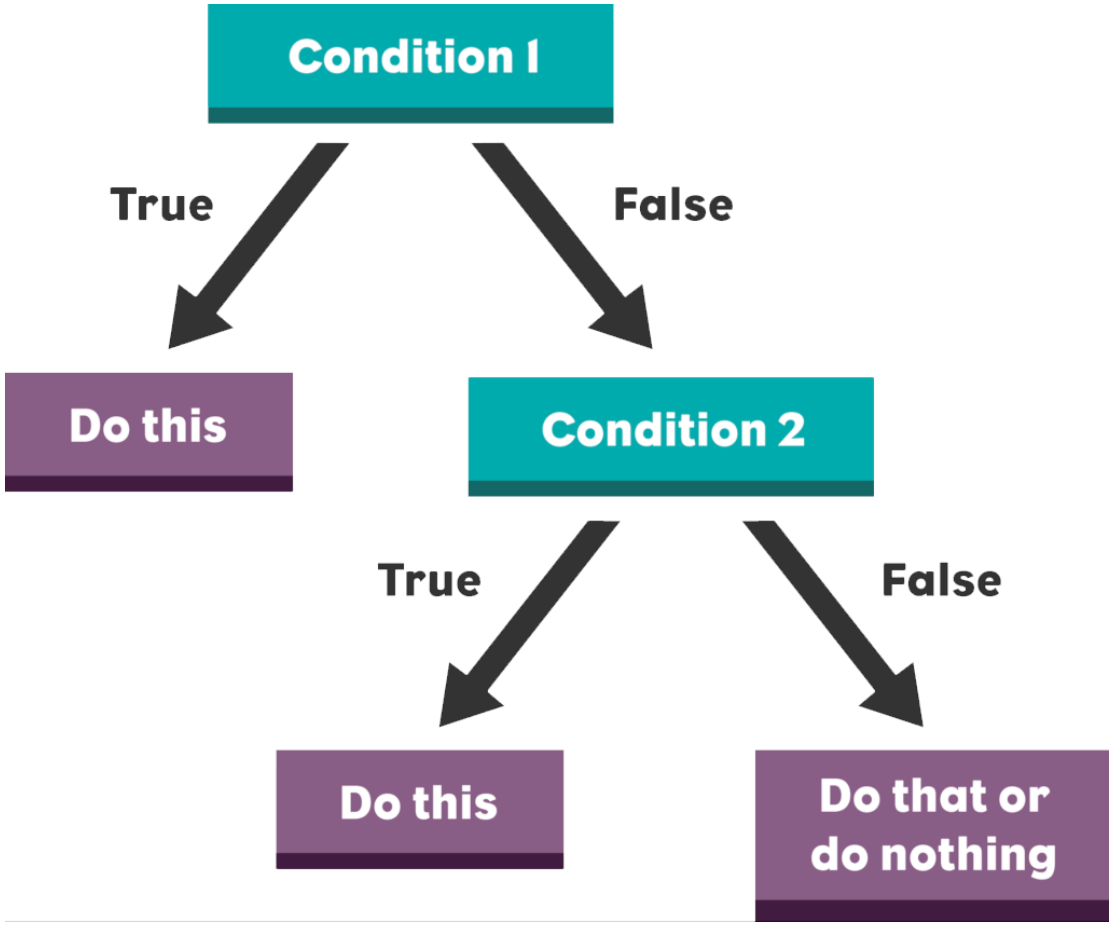
- let creates a **block-scoped** variable

```
//using let
function example() {

  if (true) {

    let count = 10;
    console.log(count);
    //Output: 10
  }

  console.log(count);
  //Output: Uncaught
  //Reference Error:
  //count is not defined
}
```

- const can be assigned only once, and its value **cannot be changed** afterwards.

```
// Using constant
const z = 10;
z = 20;

// This will result
//in an error
console.log(z);
```

Q. What are the types of conditions statements in JS? V. IMP.



Q. What are the types of conditions statements in JS? V. IMP.

Types of condition statements

1. If/ else statements

2. Ternary operator

3. Switch statement

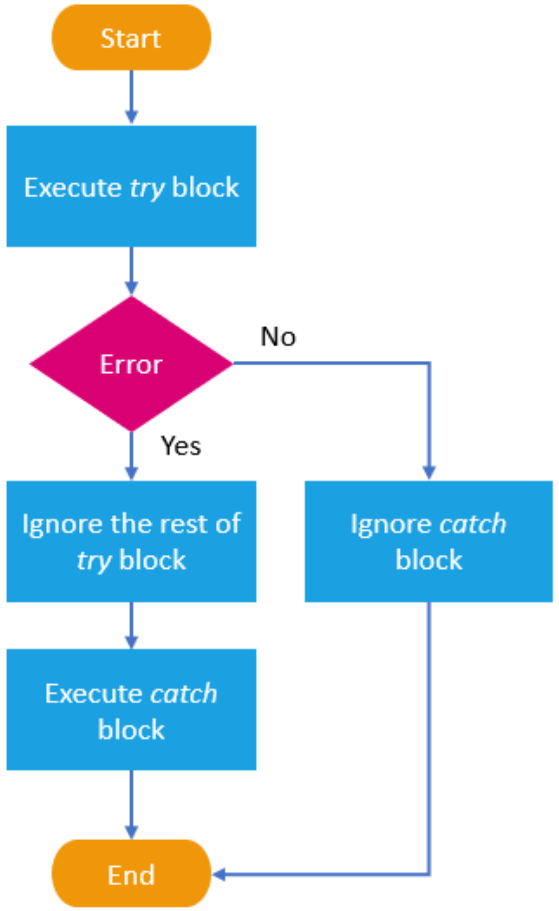
```
let x = 5;

if (x > 10) {
  console.log("1");
} else if (x < 5) {
  console.log("2");
} else {
  console.log("3");
}
// Output: '3'
```

```
let y = 20;
let z = y > 10 ? "1" : "0"
console.log(z);
// Output: '1'
```

```
let a = 5;
switch (a) {
  case 1:
    console.log("1");
    break;
  case 5:
    console.log("2");
    break;
  default:
    console.log("3");
}
// Output: '2'
```

Q. What is Error Handling in JS? V. IMP.

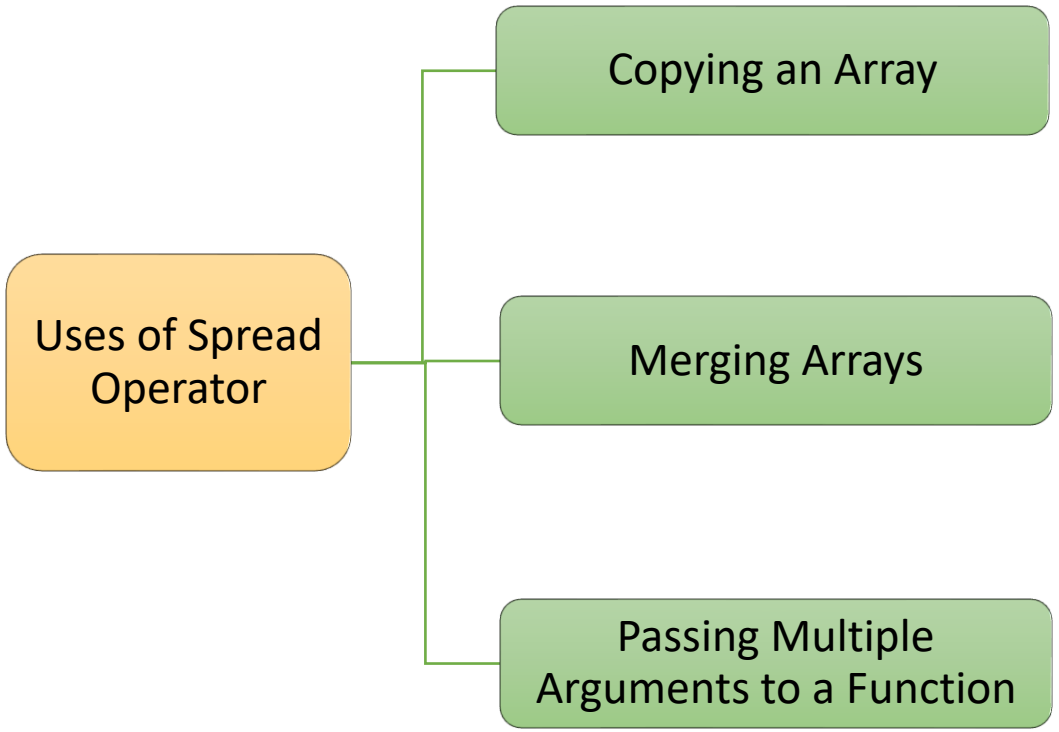
- Error handling is the process of **managing errors**.




```
//try block contains the code that might throw an error
try {

  const result = someUndefinedVariable + 10;
  console.log(result);

}
//catch block is where the error is handled
catch (error) {
  console.log('An error occurred:', error.message);
}

//Output
//An error occurred: someUndefinedVariable is not defined
```

Q. What is the difference between Spread and Rest operator in JS?

- The spread operator(...) is used to **expand or spread elements** from an iterable (such as an array, string, or object) into individual elements.




```
// Spread Operator Examples
const array = [1, 2, 3];
console.log(...array); // Output: 1, 2, 3
```

```
// Copying an array
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
console.log(copiedArray); // Output: [1, 2, 3]
```

```
// Merging arrays
const array1 = [1, 2, 3];
const array2 = [4, 5];
const mergedArray = [...array1, ...array2];
console.log(mergedArray); // Output: [1, 2, 3, 4, 5]
```

```
// Passing multiple arguments to a function
const numbers = [1, 2, 3, 4, 5];
sum(...numbers);
function sum(a, b, c, d, e) {
  console.log(a + b + c + d + e); //Output: 15
}
```

Q. What is the difference between Spread and Rest operator in JS?

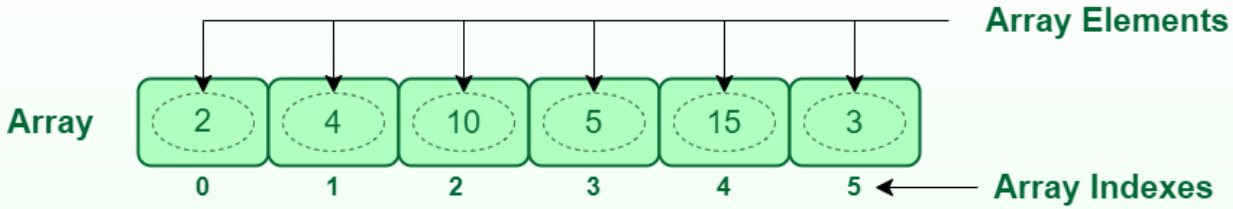
- The rest operator is used in function parameters to collect all **remaining arguments** into an array.

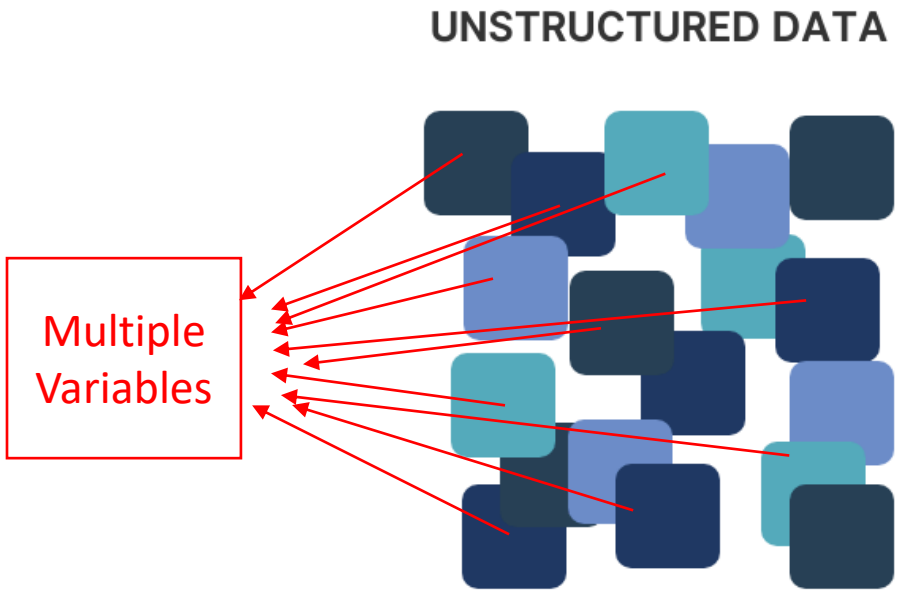
```
// Rest Operator Example
display(1, 2, 3, 4, 5);

function display(first, second, ...restArguments) {
  console.log(first); // Output: 1
  console.log(second); // Output: 2

  console.log(remaining); // Output: [3, 4, 5]
}
```

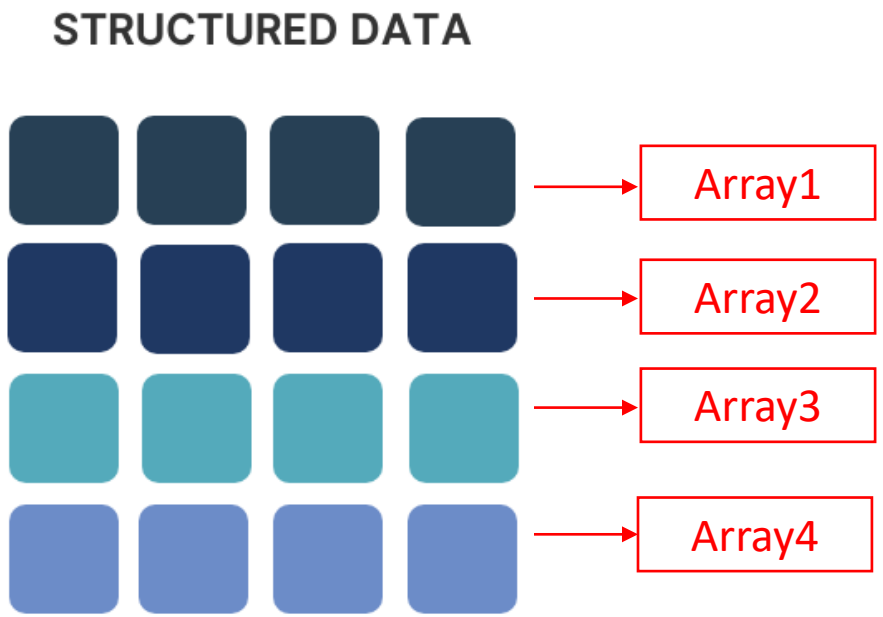
Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

- An array is a data type that allows you to **store multiple values** in a single variable.

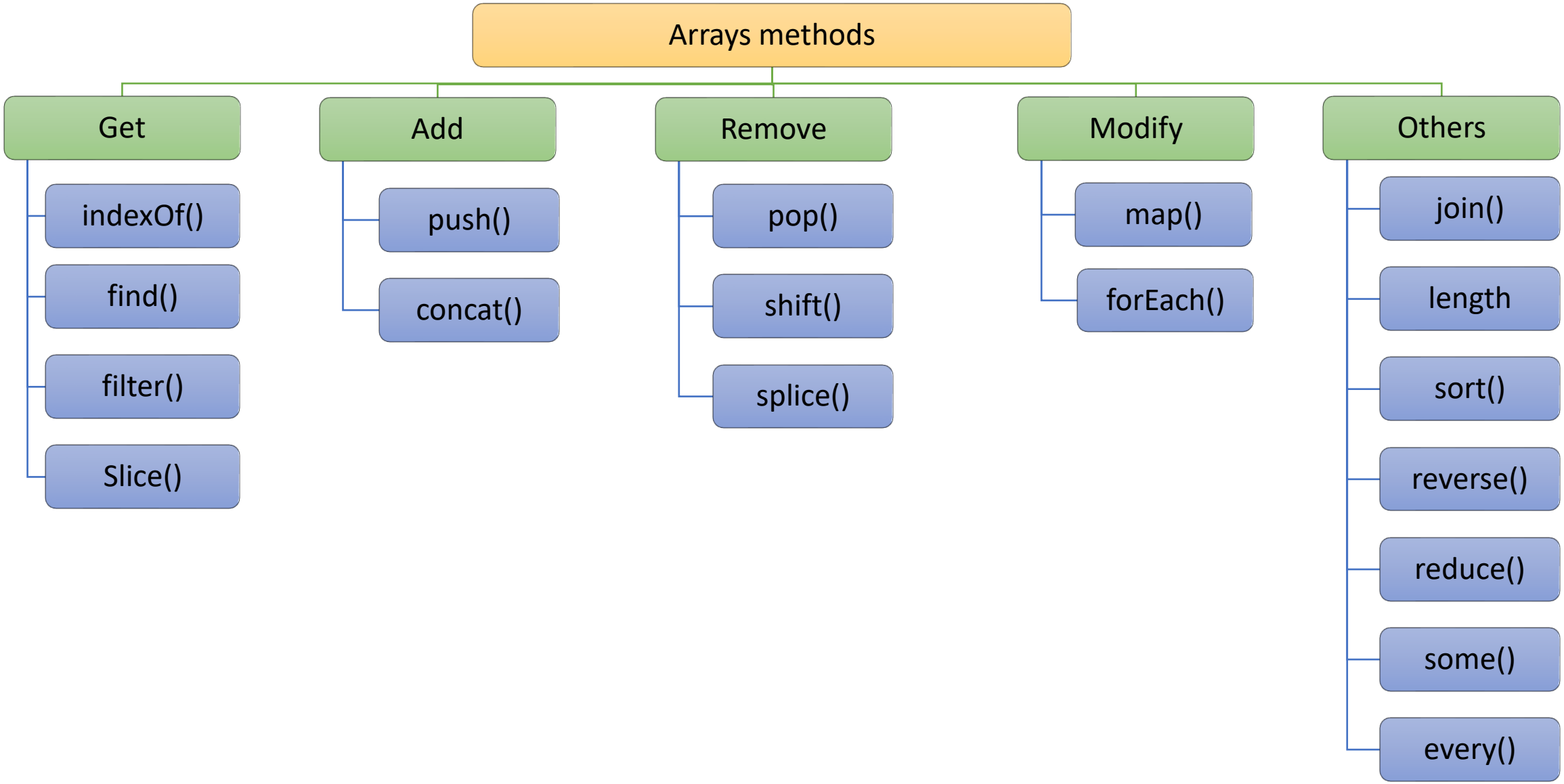
```
//Array
let fruits = ["apple", "banana", "orange"];
```







Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.



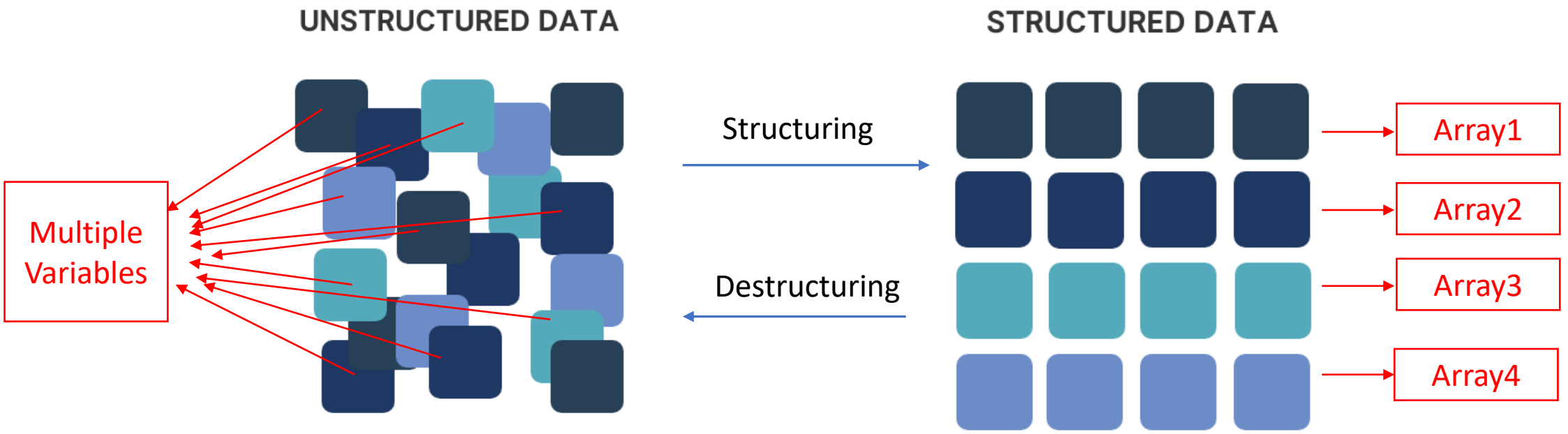
Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

❖ Pictorial representation of important method of arrays

Q. What is Array Destructuring in JS? V. IMP.

- Array destructuring allows you to extract elements from an array and assign them to **individual variables** in a single statement.

- Array destructuring is introduced in **ECMAScript 6 (ES6)**.



Q. What is Array Destructuring in JS? V. IMP.

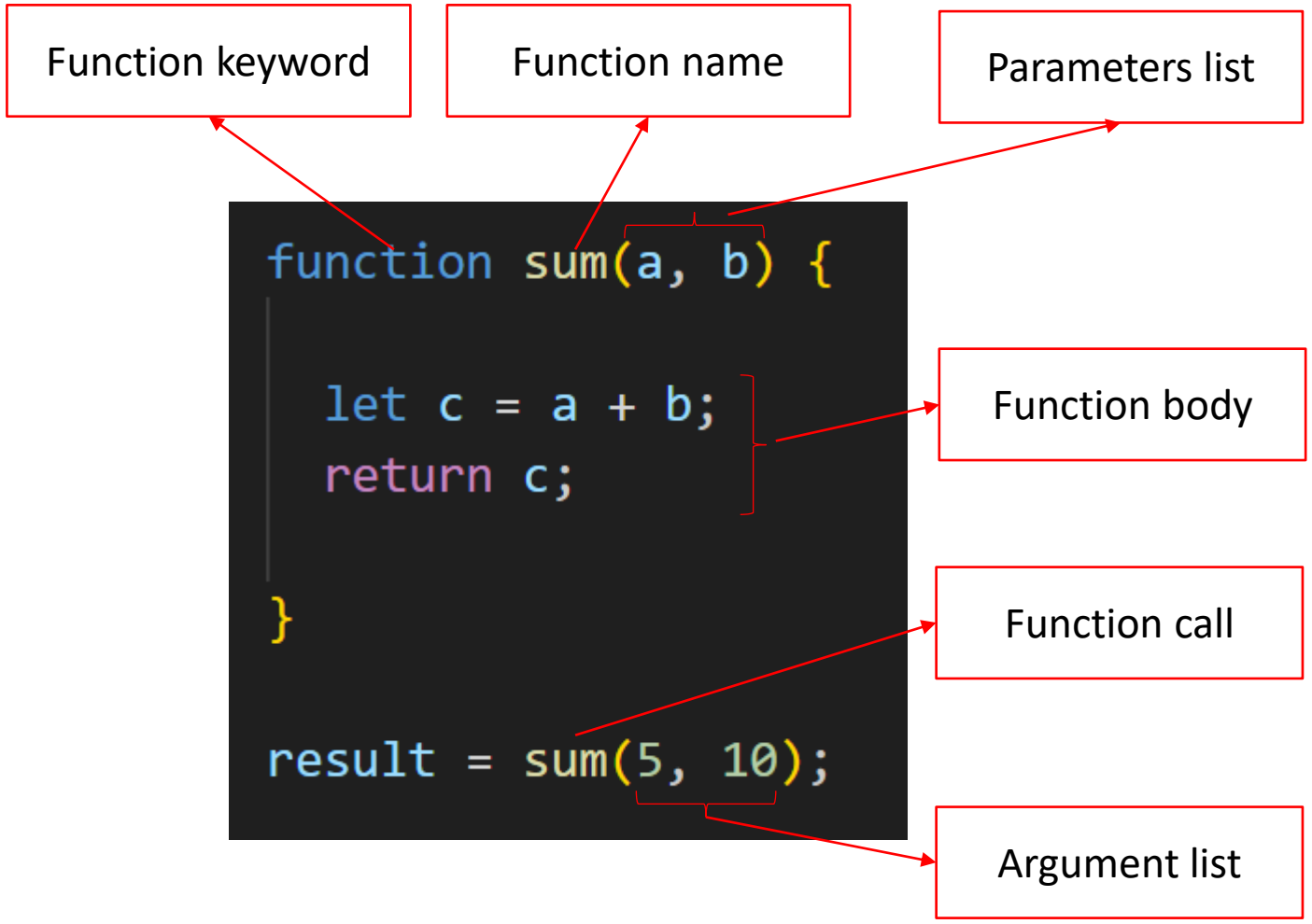
```
// Example array
const fruits = ['apple', 'banana', 'orange'];
```

```
// Array destructuring
const [firstFruit, secondFruit, thirdFruit] = fruits;
```

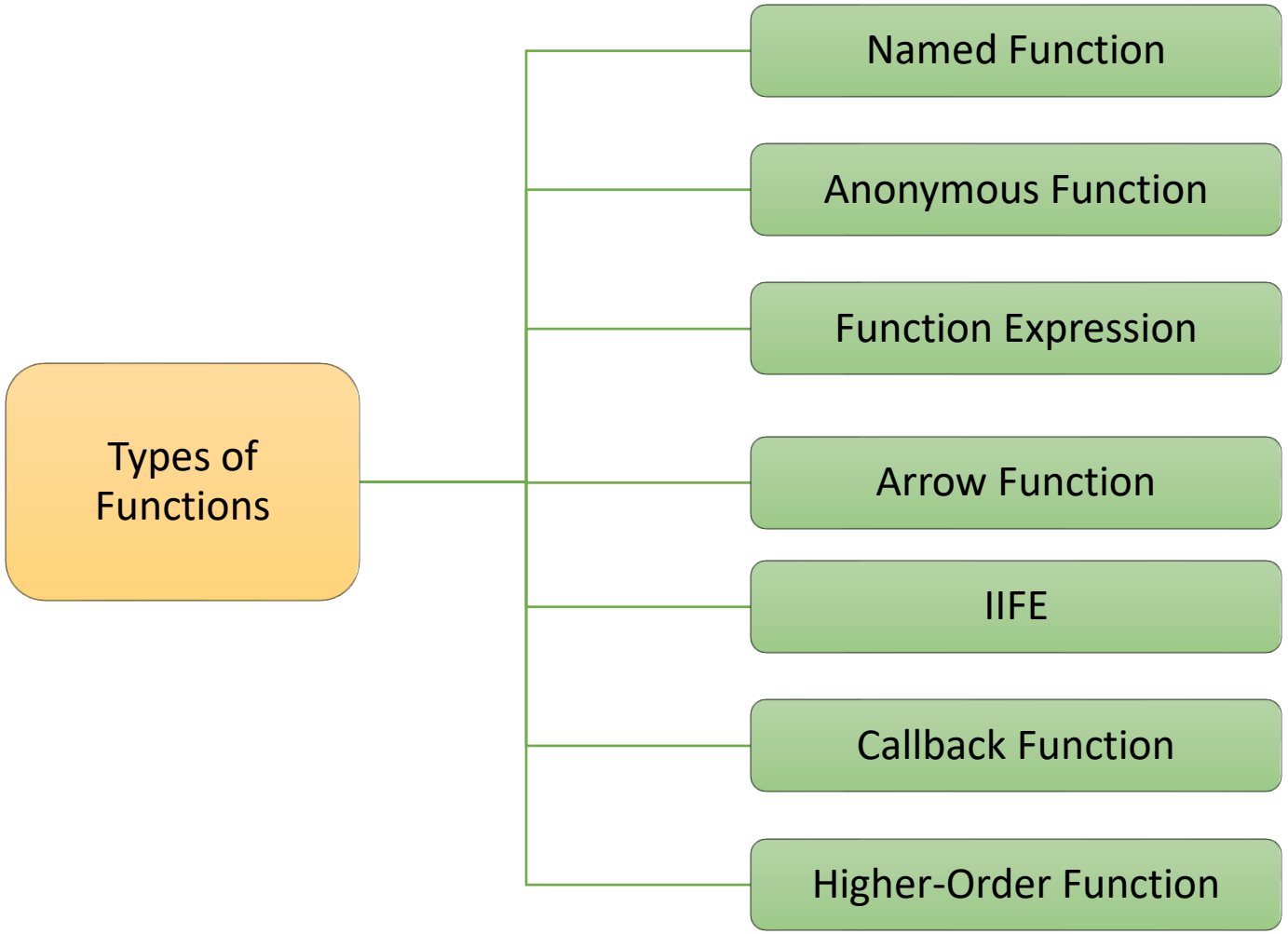
```
// Output
console.log(firstFruit); // Output: "apple"
console.log(secondFruit); // Output: "banana"
console.log(thirdFruit); // Output: "orange"
```

Q. What are Functions in JS? What are the types of function? V. IMP.

- A function is a **reusable block of code** that performs a specific task.



Q. What are Functions in JS? What are the types of function? V. IMP.



Q. What is the difference between named and anonymous functions? When to use what in applications?

- Named functions have a **name identifier**

```
//Named Function
//Function Declaration

function sum(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

- Anonymous functions **do not have a name identifier** and cannot be referenced directly by name.

```
// Anonymous function

console.log(function(a, b) {

  return a * b;

}(4, 5));

// Output: 20
```

- Use named functions for **big and complex** logics.
- Use when you want to **reuse** one function at multiple places.

- Use anonymous functions for **small logics**.
- Use when want to use a function in a **single place**.

Q. What is function expression in JS?

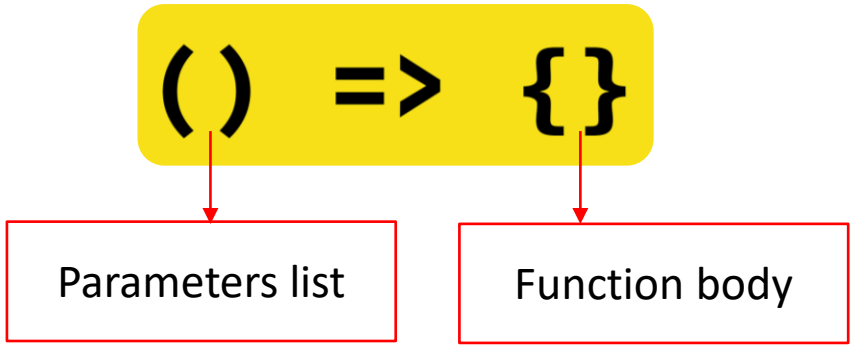
- A function expression is a way to define a function by **assigning it to a variable**.

```
//Anonymous Function Expression

const add = function(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

```
//Named Function Expression

const add = function sum(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

Q. What are Arrow Functions in JS? What is it use? V. IMP.

- Arrow functions, also known as fat arrow functions, is a **simpler and shorter** way for defining functions in JavaScript.




```
//Traditional approach

function add(x, y)
{
    return x + y;
}

console.log(add(5, 3));
//output : 8
```

```
//Arrow function

const add = (x, y) => x + y;

console.log(add(5, 3));
//output : 8
```

Q. What are Callback Functions? What is it use? V. IMP.

- A callback function is a function that is **passed as an argument** to another function.

```
function add(x, y) {
  return x + y;
}

let a = 3, b = 5;
let result = add(a, b)

console.log(result);
//Output: 8
```

Higher-order function

Callback function

```
function display(x, y, operation) {

  var result = operation(x, y);
  console.log(result);

}

display(10, 5, add);

display(10, 5, multiply);
display(10, 5, subtract);
display(10, 5, divide);
```

Q. What is Higher-order function In JS?

- A Higher order function:

1. Take one or more functions as **arguments**(callback function) OR
2. **Return** a function as a result

```
//Take one or more functions
//as arguments
function hof(func) {

  func();
}

hof(sayHello);

function sayHello() {
  console.log("Hello!");
}
// Output: "Hello!"
```

Callback function

High-order function

```
//Return a function as a result
function createAdder(number) {

  return function (value) {
    return value + number;
  };

}

const addFive = createAdder(5);

console.log(addFive(2));

// Output: 7
```

Q. What are Pure and Impure functions in JS?

1. A pure function is a function that always produces the **same output for the same input**.
2. Pure functions cannot modify the **state**.
3. Pure functions cannot have **side effects**.

```
// Pure function
function add(a, b) {
  return a + b;
}

console.log(add(3, 5));
// Output: 8

console.log(add(3, 5));
// Same Output: 8
```

1. An impure function, can produce **different outputs for the same input**.
2. Impure functions can modify the state.
3. Impure functions can have side effects.

```
// Impure function
let total = 0;

function addToTotal(value) {
    total += value;
    return total;
}
console.log(addToTotal(5));
// Output: 5

console.log(addToTotal(5));
// Not same output: 10
```

Q. What are template literals and string interpolation in strings? V. IMP.

- A template literal, also known as a template string, is a feature introduced in ECMAScript 2015 (ES6) for **string interpolation** and **multiline strings** in JavaScript.

`${Template} Literal`

```
// Backticks (`)
//Template literals with string interpolation
var myname = "Happy";
var str3 = `Hello ${myname}!`;
console.log(str3);
// Output: Hello Happy!
```

```
// Backticks (`)
//Template literals for multiline strings
var multilineStr = `
This is a
multiline string.
`;
```

Q. What are Objects in JS? V. IMP.

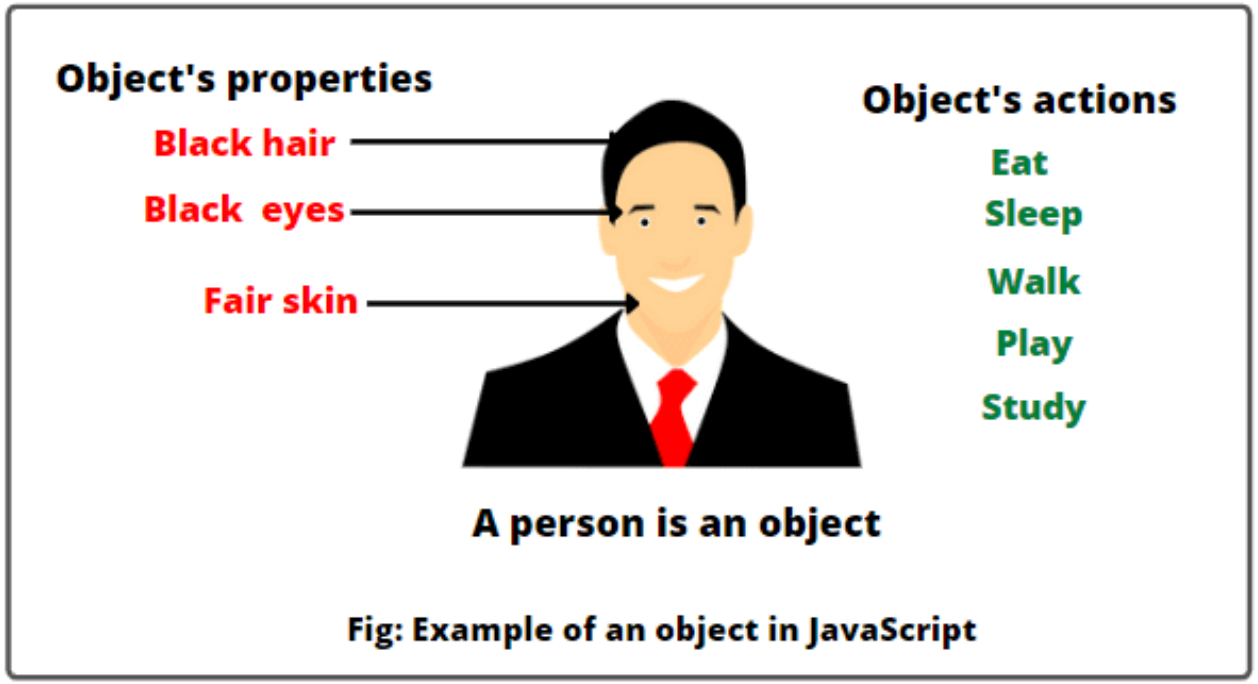
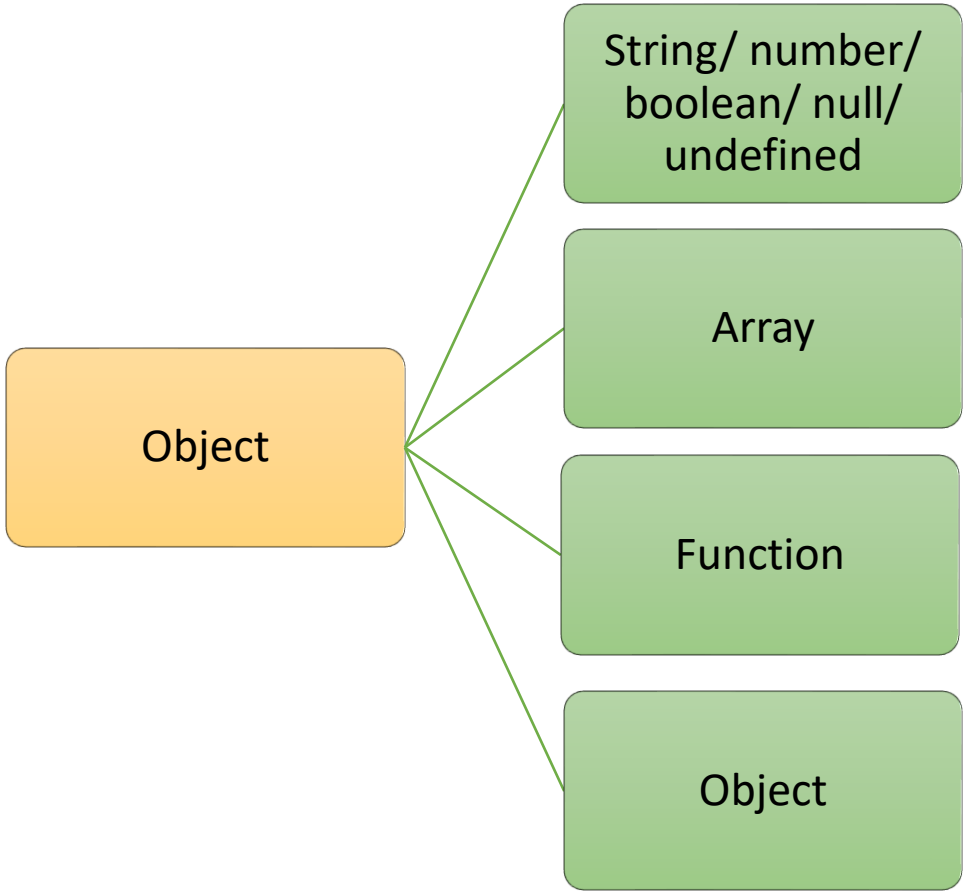


Fig: Example of an object in JavaScript

Q. What are Objects in JS? V. IMP.

- An object is a data type that allows you to store **key-value** pairs.




```
//Object Example
let person = {
  name: "Happy",
  hobbies: ["Teaching", "Football", "Coding"],
  greet: function () {
    console.log("Name: " + this.name);
  },
};

console.log(person.name);
// Output: "Happy"

console.log(person.hobbies[1]);
// Output: "Football"

person.greet();
// Output: "Name: Happy"
```

Q. What is the difference between an array and an object?

Arrays	Objects
1. Arrays are collection of values.	Objects are collections of key-value pairs.
2. Arrays are denoted by square brackets [].	Objects are denoted by curly braces {}.
3. Elements in array are ordered.	Properties in objects are unordered.

```
// Array
var fruits = ["apple", "banana", "orange"];
```

```
// Object
var person = {
  name: "Amit",
  age: 25,
  city: "Delhi"
};
```

Q. How do you add or modify or delete properties of an object?

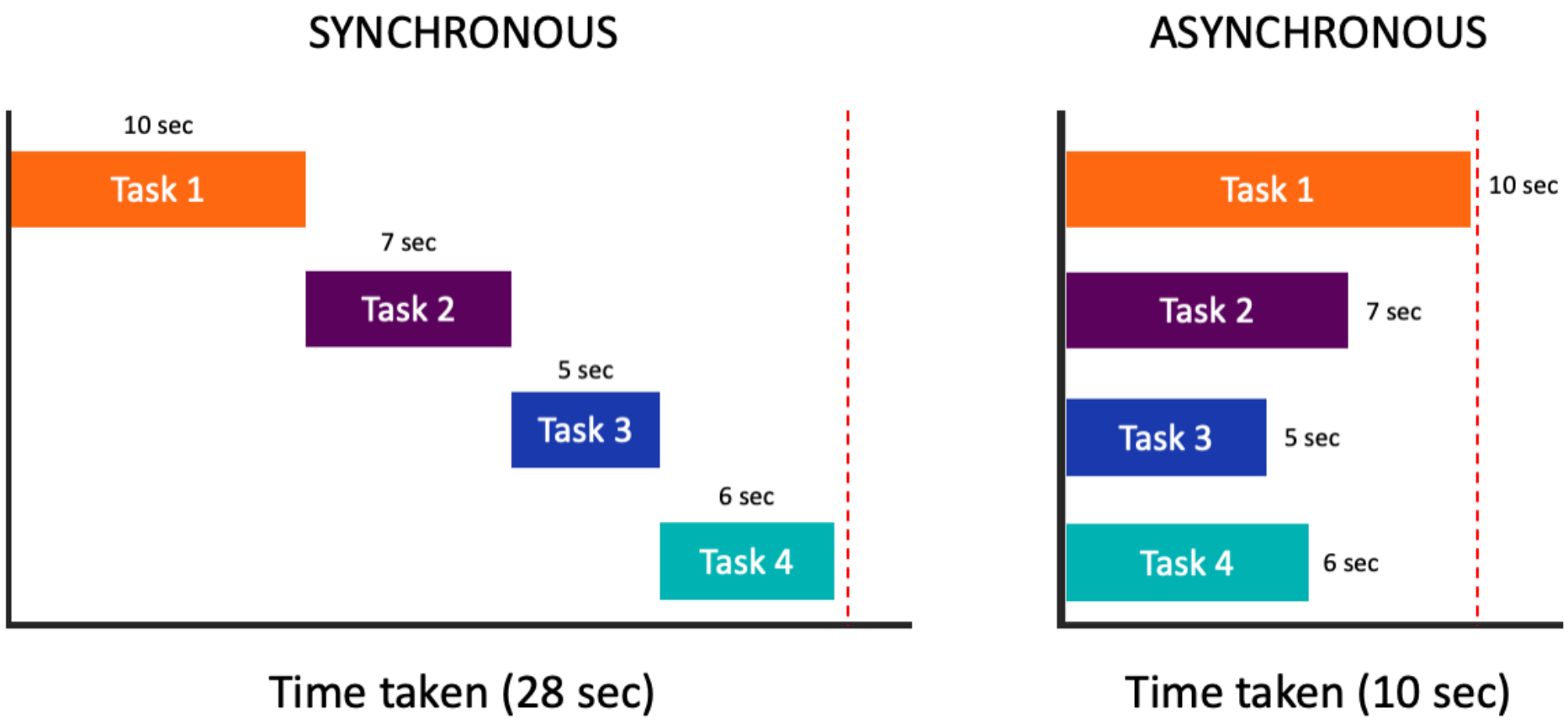
```
//Blank object
var person = {};
```

```
// Adding Properties
person.name = "Happy";
person.age = 35;
person.country = "India"
```

```
// Modifying Properties
person.age = 30;
```

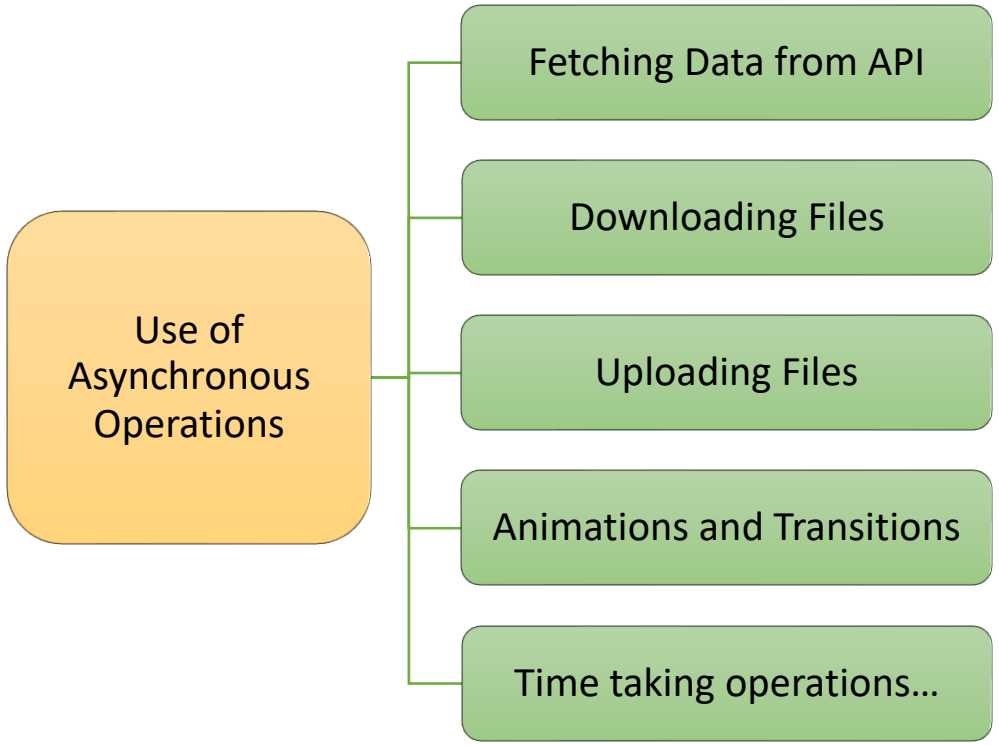
```
// Deleting Properties
delete person.age;
```

Q. What is asynchronous programming in JS? What is its use? V. IMP.



Q. What is asynchronous programming in JS? What is its use? V. IMP.

- Asynchronous programming allows multiple tasks or operations to be initiated and **executed concurrently**.
- Asynchronous operations **do not block** the execution of the code.




```
// Synchronous Programming
// Not efficient
console.log("Start");
Function1();
Function2();
console.log("End");

// Time taking function
function Function1() {
  // Loading Data from an API
  // Uploading Files
  // Animations
}
function Function2() {
  console.log(100 + 50);
}
```

Q. What is the difference between synchronous and asynchronous programming?

Synchronous programming	Asynchronous programming
1. In synchronous programming, tasks are executed one after another in a **sequential manner**.	In synchronous programming, tasks can start, run, and complete in parallel
2. Each task must complete before the program moves on to the next task.	Tasks can be executed independently of each other.
3. Execution of code is blocked until a task is finished.	Asynchronous operations are typically non-blocking.
4. Synchronous operations can lead to blocking and unresponsiveness.	It enables better concurrency and responsiveness.
5. It is the default mode of execution.	It can be achieved through techniques like callbacks, Promises, async/await

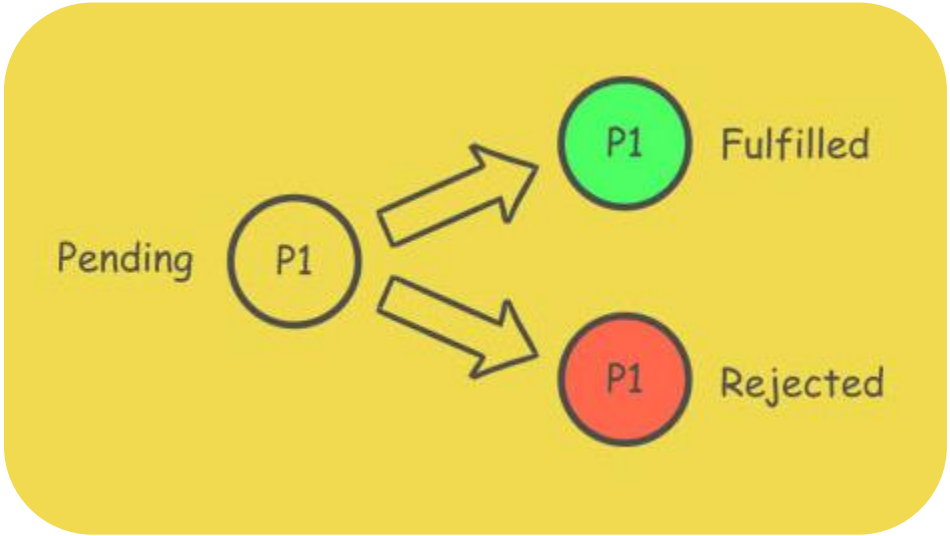
Q. What are Promises in JavaScript? V. IMP.

❖ Important points about promises:

1. Promises in JavaScript are a way to handle **asynchronous operations**.

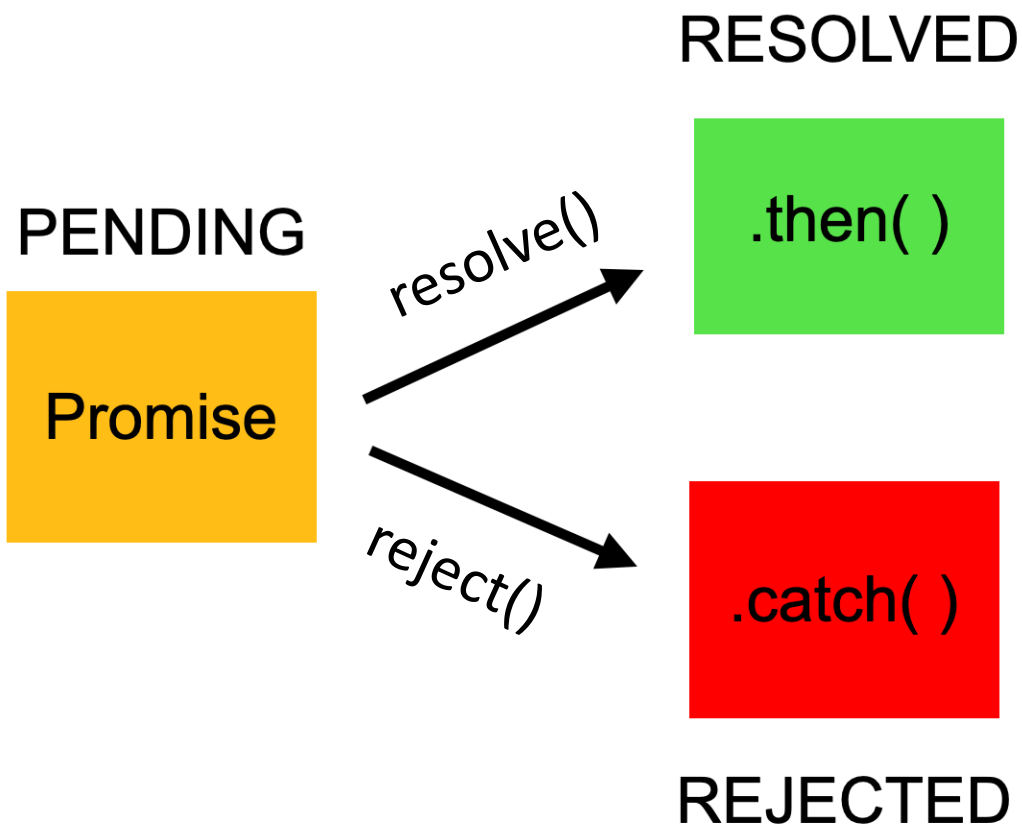
2. A Promise can be in one of three states: **pending, resolved, or rejected**.

3. A promise represents a value that **may not be available yet** but will be available at some point in the **future**.




```
const promise = new Promise((resolve, reject) => {
    // Perform asynchronous operation for eg: setTimeout()
    // Call `resolve` function when the operation succeeds
    // Call `reject` function when the operation encounters an error
});
```

Q. How to implement Promises in JavaScript? V. IMP.



Q. How to implement Promises in JavaScript? V. IMP.

```
// Create a new promise using the Promise constructor
const myPromise = new Promise((resolve, reject) => {

// Set a timeout of 1 second
setTimeout(() => {

    // Generate a random number between 0 and 9
    const randomNum = Math.floor(Math.random() * 10);

    // Resolve the promise
    if (randomNum < 5) {
      resolve(`Success! Random number: ${randomNum}`);
    }
    // Reject the promise
    else {
      reject(`Error! Random number: ${randomNum}`);
    }

}, 1000);
```

```
// Use the .then() method to
//handle the resolved promise
myPromise
  .then((result) => {
    console.log(result);
  })

  // Use the .catch() method to
  //handle the rejected promise
  .catch((error) => {
    console.error(error);
  });

//Output: Error! Random number: 9
//Output: Success! Random number: 4
```

Q. Explain the use of async and await keywords in JS? V. IMP.

- The **async keyword** is used to define a function as an **asynchronous function**, which means the code inside async function will not block the execution other code.
- The **await keyword** is used within an async function to **pause the execution** of the function until a Promise is resolved or rejected.

```
function delay(ms) {
  return new Promise((resolve) =>
    setTimeout(() => {
      console.log("Running");
      resolve();
    }, ms)
  );
}
```

```
async function greet() {
  console.log("Starting...");

  delay(2000); // Not block
  console.log("Not Blocked");

  await delay(1000); // Block the code until completion
  console.log("Blocked");
}

greet();
```

```
// Output:
// Starting...
// Not Blocked
// Running (after 1 sec)
// Blocked
// Running (after 2 sec)
```

Q. What are Classes in JS?

- Classes serve as **blueprints** for creating objects and define their structure and behavior.

- Advantages of classes:
  1. Object Creation
  2. Encapsulation
  3. Inheritance
  4. Code Reusability
  5. Polymorphism
  6. Abstraction

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}
```

```
// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

```
// Accessing properties and calling methods
console.log(person1.name); // Output: "Alice"
person2.sayHello(); // Output: "Bob - 30"
```

Q. What is a constructor?

- Constructors are special methods within classes that are **automatically called** when an object is created of the class using the new keyword.

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}

// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

Q. What is the use of this keyword?

- this keyword provides a way to access the **current object or class**.

```
// class example
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(`${this.name}`);
  }
}

var person1 = new Person("Happy")
console.log(person1.name);
```

```
// constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

21: Typescript

Q1. What is Typescript? Or What is the difference between Typescript and Javascript?

Q2. How to install Typescript and check version?

Q3. What is the difference between let and var keyword?

Q4. What is Type annotation?

Q5. What are Built in/ Primitive and User-Defined/ Non-primitive Types in Typescript?

Q6. What is “any” type in Typescript?

Q7. What is Enum type in Typescript?

Q8. What is the difference between void and never types in Typescript?

Q9. What is Type Assertion in Typescript?

Q10. What are Arrow Functions in Typescript?

21: Typescript

Q11. What is Object Oriented Programming in Typescript?

Q12. What are Classes and Objects in Typescript?

Q13. What is Constructor?

Q14. What are Access Modifiers in Typescript?

Q15. What is Encapsulation in Typescript?

Q16. What is Inheritance in Typescript?

Q17. What is Polymorphism in Typescript?

Q18. What is Interface in Typescript?

Q19. What's the difference between extends and implements in TypeScript ?

Q20. Is Multiple Inheritance possible in Typescript?

Q55

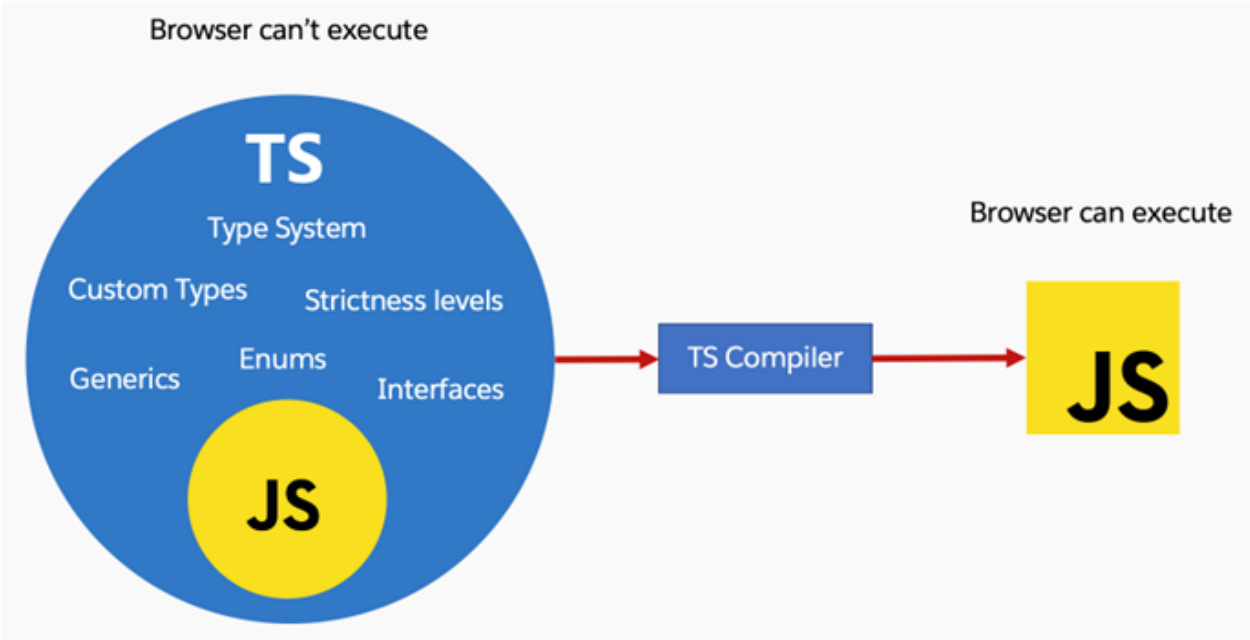
What is Typescript? What are the advantages of Typescript over Javascript? V. IMP.

TS

- Typescript is an open-source programming language.
  It's advantage are:

1. Typescript is a strongly typed language.
2. Typescript is a superset of JavaScript.
3. It has Object oriented features.
4. Detect error at compile time.

- In image, you can see TS(Typescript) contains JS(Javascript) and other additional features(type system, custom types, generics, enums, interfaces).
- Browser can't execute typescript, so finally TS Compiler will convert the TS to JS only, which browser can understand.

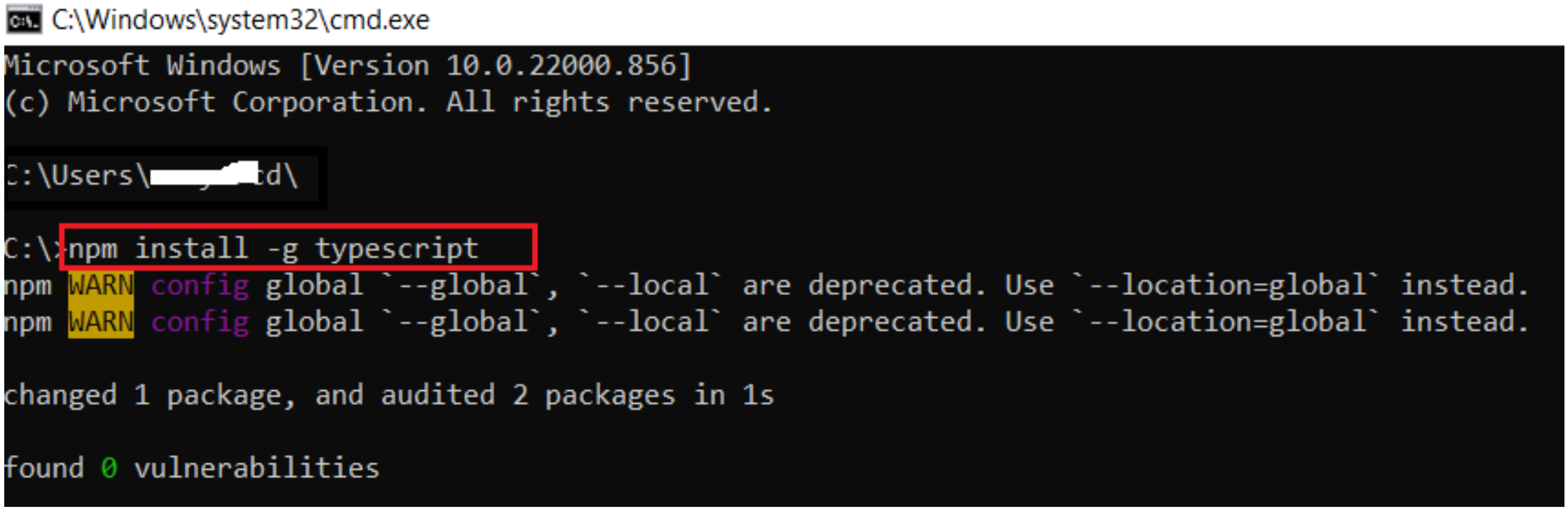


Q56

How to install Typescript and check version?

TS

- Here is the command which can be used to install typescript on your system(without Angular).



Q57

What is the difference between let and var keyword?

TS

- var and let are both used for variable declaration, but the difference between them is that **var is global function scoped and let is block scoped**.

```
function fnLetVar()
{
    //let keyword
    for(let i=0; i<5; i++)
    {
        console.log("Inside " + i);
    }
    console.log("Outside " + i);

    //var keyword
    for(var j=0; j<5; j++)
    {
        console.log("Inside " + j);
    }
    console.log("Outside " + j);
}

fnLetVar();
```

Compile time error, since let variable “i” scope is only limited inside for loop.

No error, since var variable “j” scope is not limited inside for loop, but inside the whole function fnLetVar().

Q58

What is Type annotation in Typescript?

TS

- Type annotations helps the compiler in checking the types of variable and avoid errors.

For example, in below image when we set variable “i” as number. And now if you assign “i” a string or Boolean value, then Typescript will show the compile time error, which is good.

```
let i : number;

i = 1;

i = "Happy";

i = true;
```

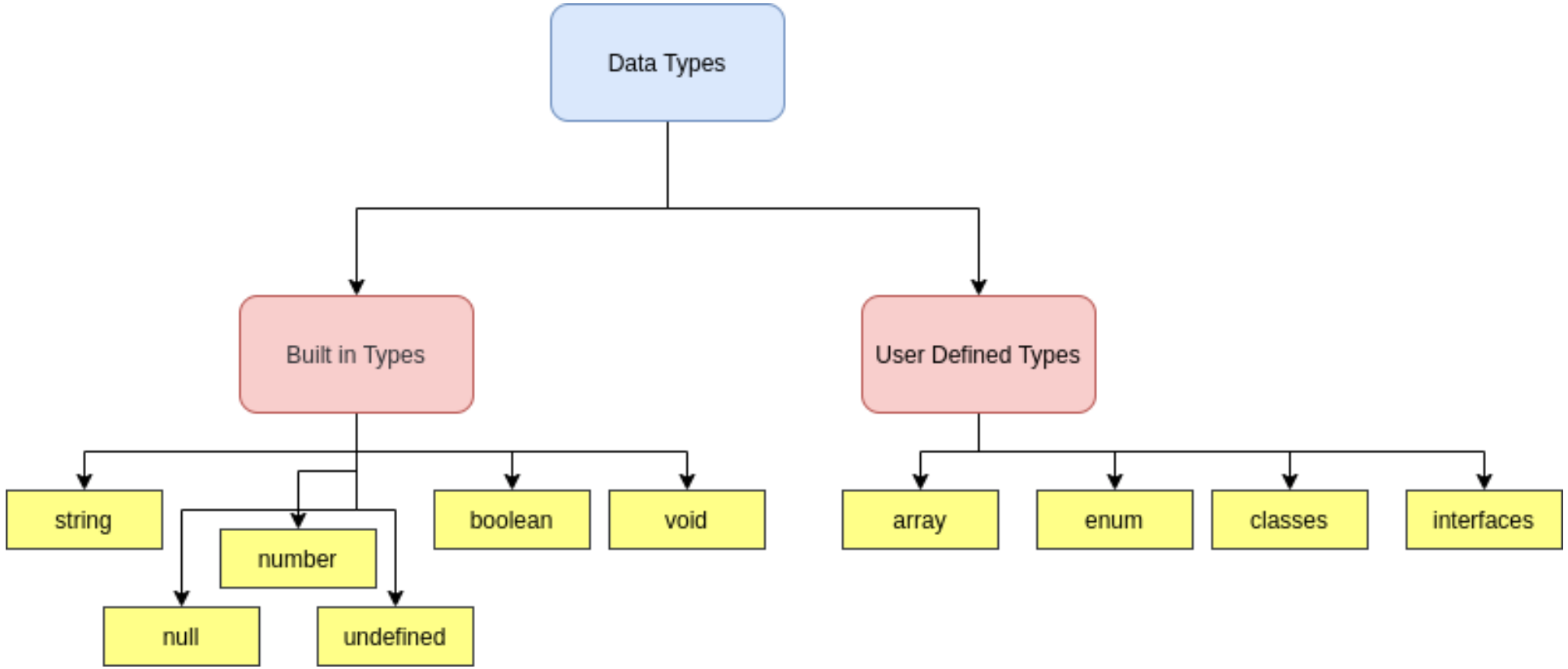
Q59

What are Built in/ Primitive and User-Defined/ Non-primitive Types in Typescript?

TS

- Built-in Data types are used to store simple data. Such as string, number, Boolean.

- User-Defined types are used to store complex data. Such as array, enum, classes.



Q60

What is “any” type in Typescript? V. IMP.

TS

- When a value is of type any, then **no typechecking(no compile time errror)** will be done by compiler and the flexibility is there to do anything with this variable.

For example, see in the image there is no compile time error for assigning different types to variable “x” because it is of any type.

```
let x: any;

x = 1;

x = "Happy";

x = true;

x();

x.AnyProperty = 10;
```

Q61

What is Enum type in Typescript?

TS

- Enums allows to define a set of **named constants**.

Suppose you have a family of constants(Directions) like this in image.
Now if you want to insert some direction, for example SouthWest between 1 and 2, then it will be a problem, as you have to change this in all of your application.

```
const DirectionNorth = 0;
const DirectionSouth = 1;
const DirectionWest = 2;
const DirectionEast = 3;

let dir = DirectionEast;
```

So here enum is a better way as you don't have to assign the numbers then.

```
enum Direction {
    North,
    South,
    West,
    East
}

let dirNew = Direction.East;
```

Q62

What is the difference between void and never types in Typescript?

TS

- void means no type. It is used when the function will return empty.

In image, fnVoid() function will return empty.

```
let message = "Happy";

function fnVoid(message: string): void {

    console.log("void " + message);
}
```

- never means it will return never. It is used to throw error only.

For example, in image fnNever function which will not reach to its last line, and it always throws an exception.

```
function fnNever(message: string): never {

    throw new Error(message);

}
```

Q63

What is Type Assertion in Typescript?

TS

- Type assertion is a technique that informs the compiler about the type of a variable.

For example, in below image, if we do not put <String>, then typescript will try to assume the type of "secondname" by itself automatically.
But by putting <String>, we are asserting and informing the compiler about the type of secondName as String and now compiler will not do automatic typechecking.

```
let myname: any = "Happy";

// Conversion from any to string
let secondName = <String> myname;

console.log(secondName);
console.log(typeof secondName);
```

Q64

What are Arrow Functions in Typescript?

V. IMP.

TS

- An **arrow function** expression is a compact alternative to a traditional function expression.

In below image, you can see how the same function can be written using arrow function.

```
//Normal function approach
let x = function(a, b)
{
    a * b;
}

//Arrow function approach
let y = (a, b) =>  a * b;
```

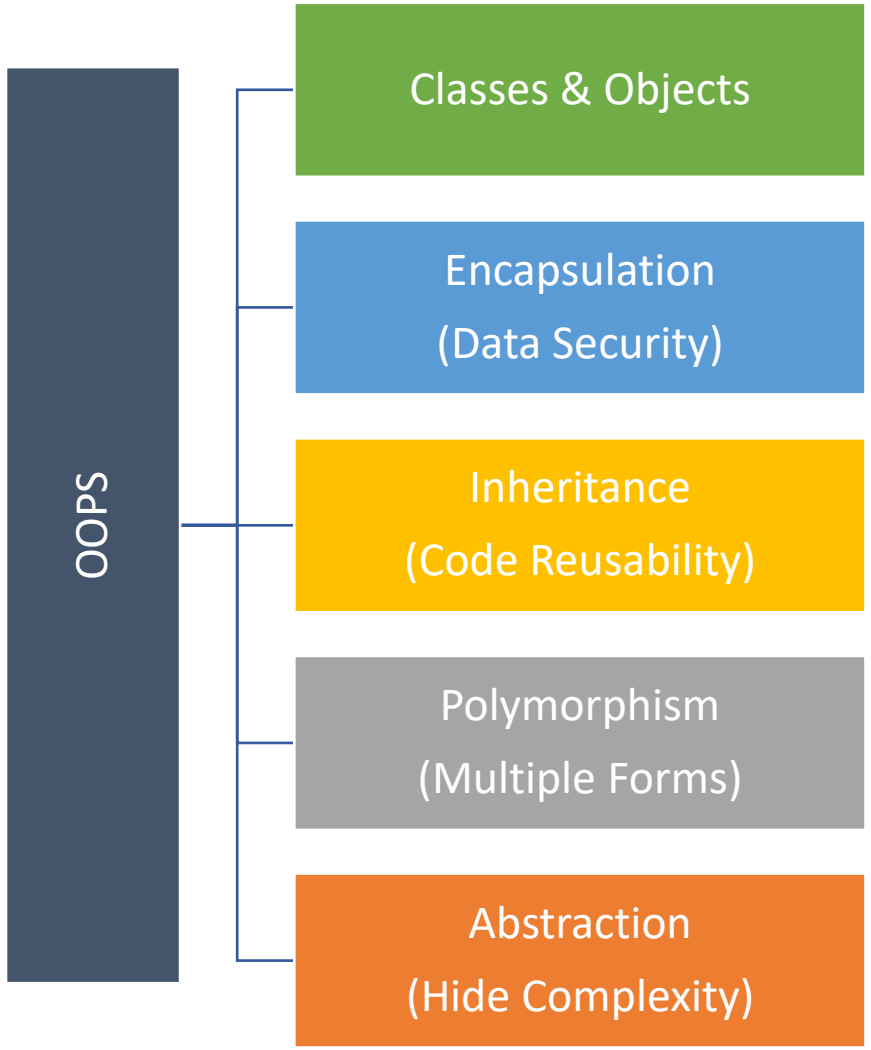
Q65

What is Object Oriented Programming(OOPS) in Typescript? V. IMP.

TS

- Object oriented programming is used to design structured and better applications.

In the image, you can see the concepts used in object oriented programming.



Q66

What are Classes and Objects in Typescript?

TS

- Classes are blueprint for individual objects.
- Objects are instances of a class.

Person is a class with two properties(name, age) and one method(familyCount)

objPerson is the object/instance of Person class. By this object only we can set Person class properties and call the method of the class.

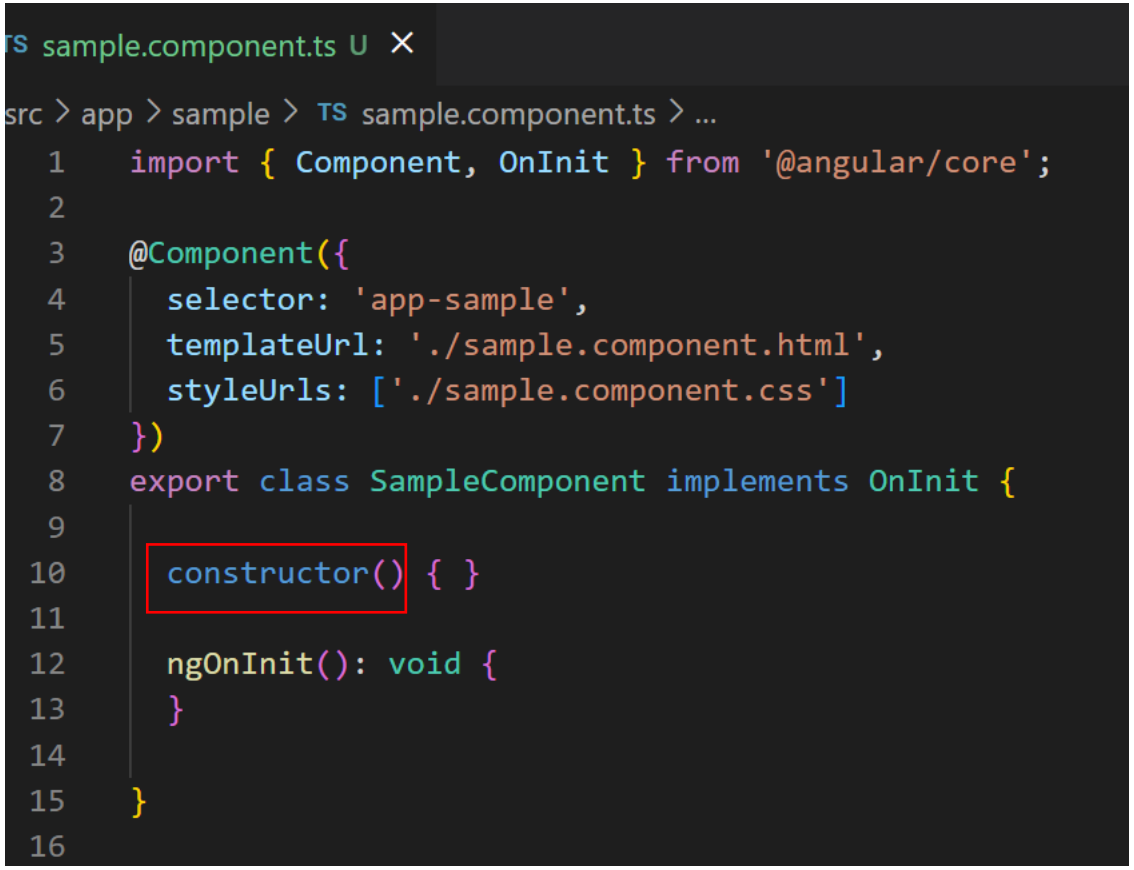
```
class Person {
    name: string;
    age: number;

    familyCount(){
      return 6;
    }
}

var objPerson = new Person();

let myName = objPerson.name = "Happy";
let myAge = objPerson.age = 10;

let countFamily = objPerson.familyCount();
```

Q67

What is a Constructor in Angular?

TS

- The constructor is a method in a **TypeScript class** that automatically gets called when the class is being **instantiated**.
- Constructor always run before any angular hook and it is not a part of Lifecycle Hooks.
- Constructor is widely used to inject **dependencies(services)** into the component class.



Q68

What are Access Modifiers in Typescript?

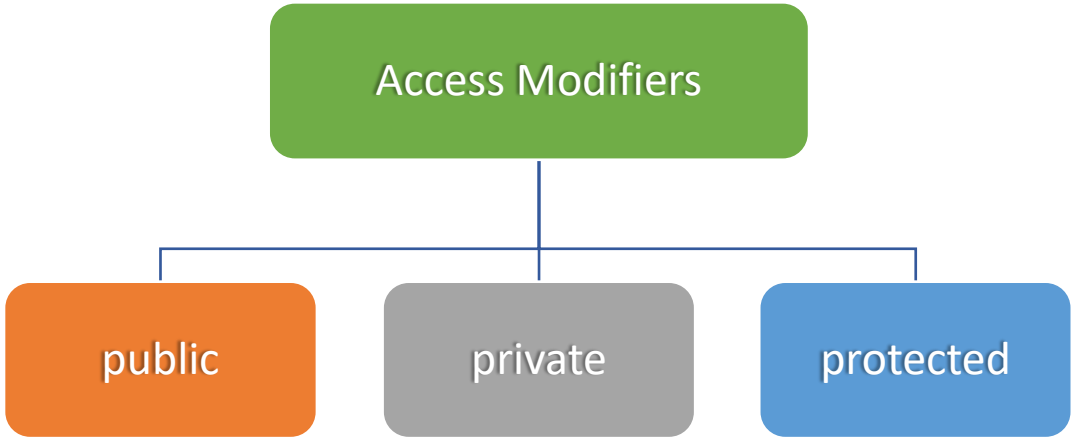
TS

❖ 3 types of Access modifiers in Typescript:

1. The public modifier allows class varibale, properties and methods to be accessible from all locations.

2. The private modifier limits the visibility to the same-class only.

3. The protected modifier allows properties and methods of a class to be accessible within same class and within subclasses/derived/child.



Q69

What are Access Modifiers in Typescript?

TS

1. empId no error, as it is public.

2. empName error, as it is out of the class and it is private.

3. empAge error, as it is out of the class and it is protected.

3. But in PermanentEmployee empAge no error, because it is inside the derived class(PermanentEmployee) of Employee.

```
class Employee
{
    public empId: number;
    private empName: string;
    protected empAge: number;
}

var objEmployee = new Employee();
objEmployee.empId = 10;
objEmployee.empName = "Amit";
objEmployee.empAge = 100;

class PermanentEmployee extends Employee
{
    empId = 100;
    empAge = 30;
}
```

Q70

What is Encapsulation in Typescript?

TS

- Encapsulation is the wrapping up of data and function together to access the data.

For example, here in code _empId is the data. We can make it public and then it can be accessible outside the Employee class.

But as per OOPS concept encapsulation this is wrong as it can lead to data security issues.

Therefore, we should mark this _empId data as private. And then create a function(getEmpId), which will help in accessing this _empId outside of this class.

Outside the class, in the last line we are calling getEmpId method to access/dispaly the data _empId.

```
class Employee
{
    private _empId: number; //data

    getEmpId(){
        return this._empId;
    }
}

let objEmployee = new Employee();

console.log(objEmployee.getEmpId());
```

- What is the advantage of Encapsulation?

This is good for data security to prevent direct data access.

Q71

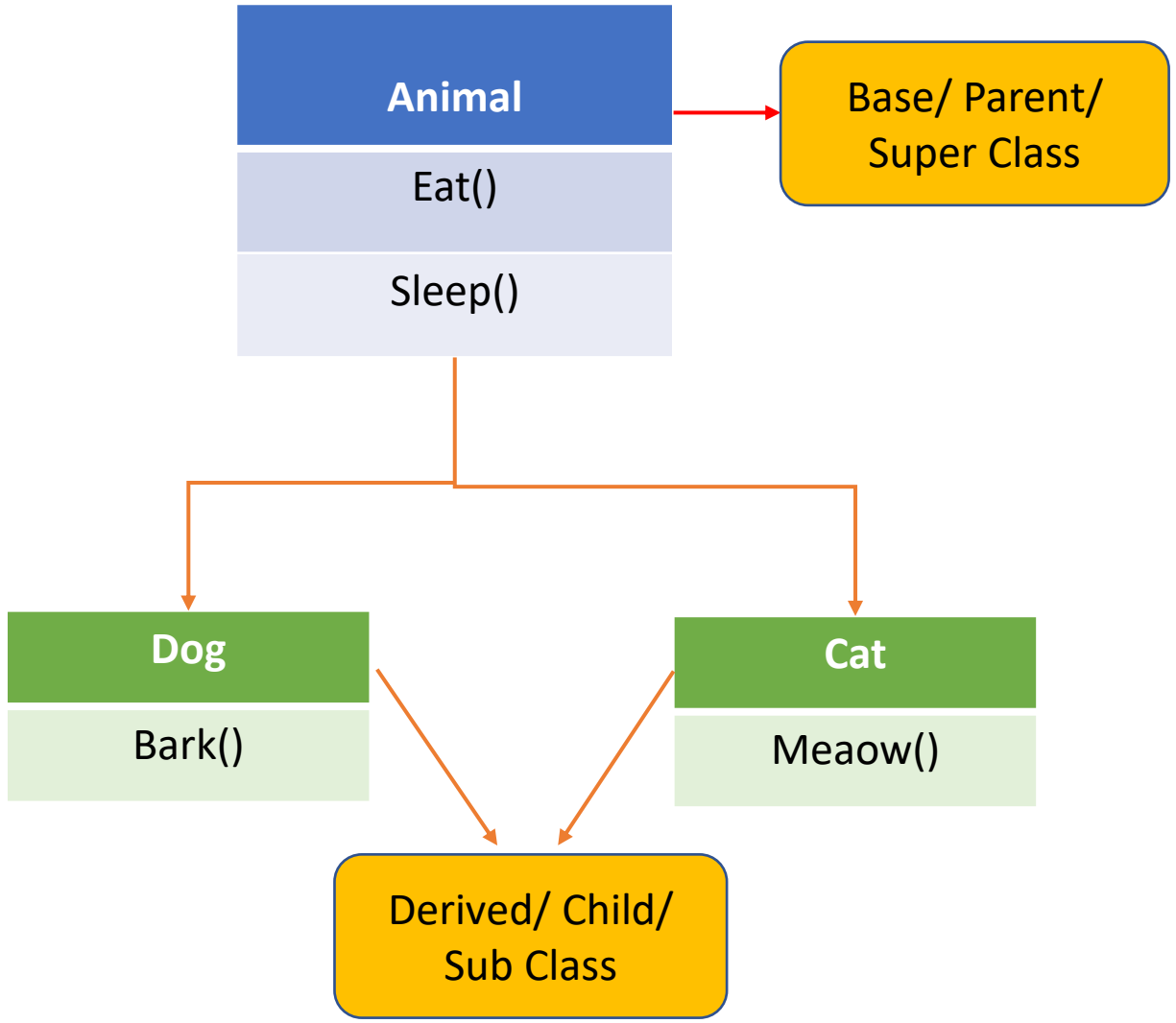
What is Inheritance in Typescript?

TS

- Inheritance is a feature where derive class will acquire the properties and methods of base class.

For example, in the image Dog and Cat class are derived from Animal base class.

Therefore, Eat() and Sleep() methods of Animal base class will be automatically available inside Dog and Cat derived class without even writing it.



- What is the advantage of Inheritance?

Inheritance is good for reusability of code. As you can write one method in base class, and then use it in multiple derived classes.

Q71

What is Inheritance in Typescript?

TS

```
class Animal{
    Eat(){
        console.log("eat");
    }
}

class Dog extends Animal{
    Bark(){
        console.log("bark");
    }
}

let objectDog = new Dog();

objectDog.Eat();

objectDog.Bark();
```

Both Eat() and Bark() method are accessible via Dog class object. Eat is from the base class but.

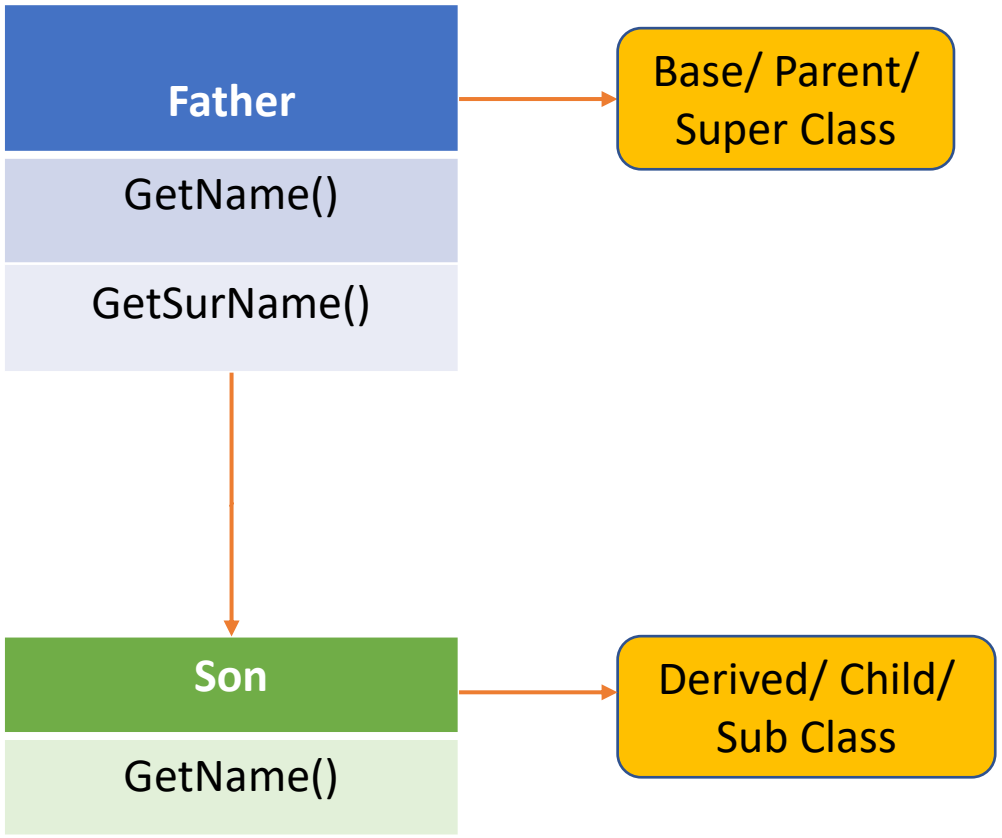
Q72

What is Polymorphism in Typescript?

TS

- Polymorphism means same name method can take multiple Forms.

For example, in the image Son can inherit the GetSurName() method implementation of father. But the GetName() method implementation should be different for son. So son can have a different implementation but the method name can be same as GetName().



- What is the advantage of Polymorphism?

Like in dictionary, a word "Running" can have multiple meanings: Running a race or Running a computer.

Similarly in programming, Polymorphism can be used to do multiple things with the same name.

Q72

What is Polymorphism in Typescript?

TS

GetName() method of Father class object output is different, and Getname() method of Son class object output is different. Same name method taking multiple forms.

```typescript
class Father {
    GetName() {
        console.log('Amit');
    }
    GetSurName() {
        console.log('Singh');
    }
}

class Son extends Father {
    GetName() {
        console.log('Ravi');
    }
}
let objectFather = new Father();
objectFather.GetName();
//Output: Amit

let objectSon = new Son();
objectSon.GetName();
//Output: Ravi
```

Q72

What is Interface and why to use them in Typescript? V. IMP.

TS

- An interface is a contract where methods are only declared.
- What is the advantage of creating Interface?

For example, we have multiple classes PermanentEmployee, ContractEmployee etc.
They all must have the Role() method, but the body of Role() method can be different.
Now to maintain consistency we will create the Interface IEmployee, with just declaration of Role() method and implement this interface in all the classes.
This will make sure all the Employees classes are in sync with the Role method. It will also help in unit testing.

- One advantage is, multiple inheritance can be achieved via interfaces only.
- Also, if we create interfaces for the classes in application then unit testing is very easy, otherwise it is difficult.

```
interface IEmployee
{
    Role(): void;
}

class PermanentEmployee implements IEmployee{

    Role(){
        console.log("Lead");
    }
}

class ContractEmployee implements IEmployee{

    Role(){
        console.log("Developer");
    }
}
```

Q73

What's the difference between extends and implements? V. IMP.

TS

- **Extends** used for base class inheritance.

```
//Parent Class
class Employee{
    Salary()
    {
        console.log("salary");
    }
}

//Child Class
class PermanentEmployee extends Employee {

}
```

- **Implements** used for interface inheritance.

```
interface IEmployee{

    Salary();

}

class PermanentEmployee implements IEmployee {
    Salary() {
        {
            console.log("50000");
        }
    }
}
```

Q74

Is Multiple Inheritance possible in Typescript?

TS

- Multiple inheritance means when one derived class is inherited from multiple base classes.

- Multiple inheritance in typescript only possible via Interfaces.

See in first image there is the compile time error in Employee2 as multiple base classes are not allowed, but in second image with multiple interfaces iit is working.

```
//Parent Class 1
class Employee1{
    Salary1()
    {
        console.log("salary");
    }
}

//Parent Class 2
class Employee2{
    Salary2()
    {
        console.log("salary");
    }
}

//Child Class
class PEmployee extends Employee1, Employee2 {

}
```

```
interface IEmployee1{

    Salary1();
}
interface IEmployee2{

    Salary2();

}

class TEmployee implements IEmployee1, IEmployee2 {
    Salary1() {
        console.log("300000");
    }
    Salary2() {
            console.log("500000");
    }
}
```


Your fate is in your hands, not in the hands of your job interviewer.
Good Luck.