# React JS

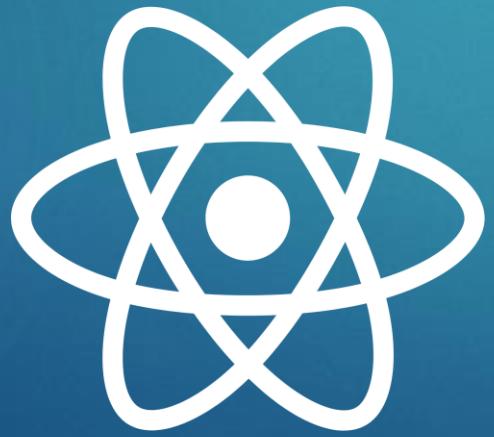
BY CODE EATER

# React JS

- React.js, commonly referred to as React, is an open-source JavaScript library developed and maintained by Facebook.
- It is a popular tool for building user interfaces (UIs) and is widely used in web development.

# JS vs React JS

Imperative Style



Declarative Style

# JS vs React JS

```javascript
const button = document.getElementById('myButton');
const paragraph = document.getElementById('myParagraph');
let count = 0;

button.addEventListener('click', () => {
  count++;
  paragraph.textContent = 'Count: ' + count;
});
```

JavaScript

```jsx
function App() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

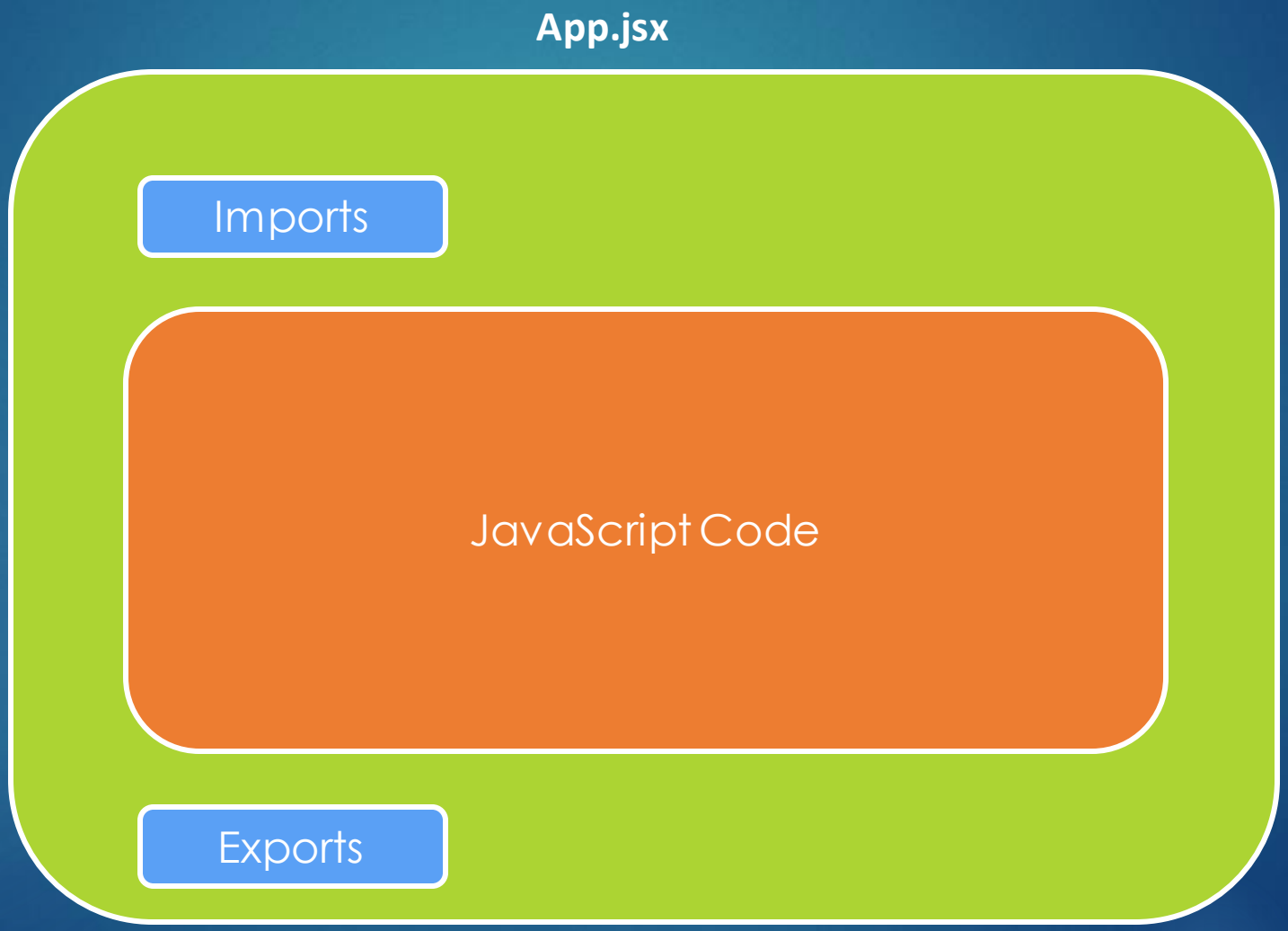
React JS

# React JS – Component Based

In React, components are the building blocks of user interfaces. They are reusable, self-contained pieces of code that represent a part of a user interface.

# React JS – Code Structure

# React JS – Code Structure



# React JS – Installation

- **Create React App** - Create React App is a tool developed by Facebook that abstracts away the build configuration and setup process.

- **Vite** - Vite is a build tool and development server for modern web development, primarily aimed at building web applications using JavaScript or TypeScript.

# React JS Flow

- Folder Structure
- Create a component
- Return and JSX
- Multiple Components
- Component Reuse
- Dynamic Data In JSX

# JSX

- JSX stands for "JavaScript XML." It is a syntax extension for JavaScript often used with libraries like React to describe the structure and content of user interfaces.

- JSX allows developers to write HTML-like code within their JavaScript code, making it easier to create and manipulate the user interface elements of a web application.

# JSX Example

Let's see an example

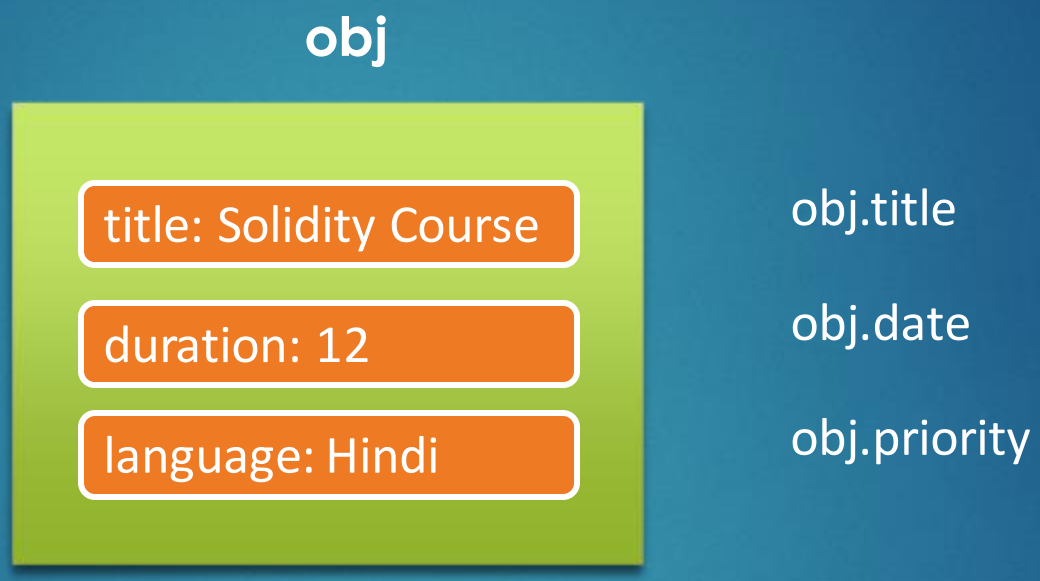
```jsx
const element = <h1>Hello, JSX!</h1>;
```

```jsx
const element = React.createElement('h1', null, 'Hello, JSX!');
```

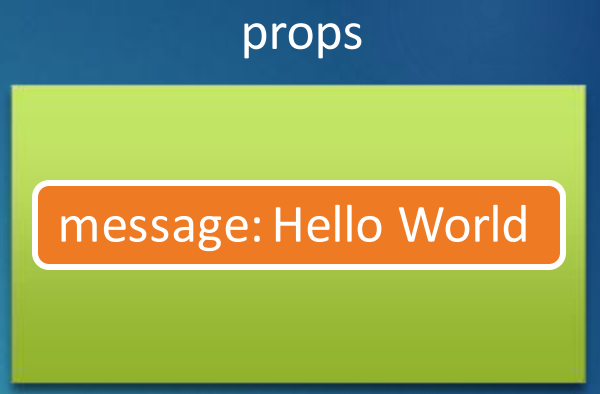
# Props

- In React.js, "props" is short for "properties," and it's a fundamental concept used to pass data from one component to another.
- Props in React are represented as **JavaScript objects**. These objects contain key-value pairs, where the keys are the names of the props
- Props are a way to make components more dynamic and reusable by allowing you to provide values or configuration to a component when it's created.
- Think props as component(function) arguments.

# Props


obj
title: Solidity Course
duration: 12
language: Hindi
obj.title
obj.date
obj.priority

# Props



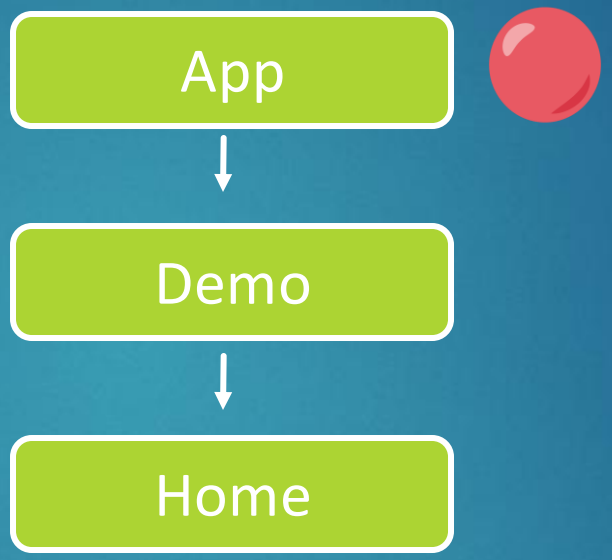

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const data = 'Hello, World!';
  return (
    <ChildComponent message={data} />
  );
}
```

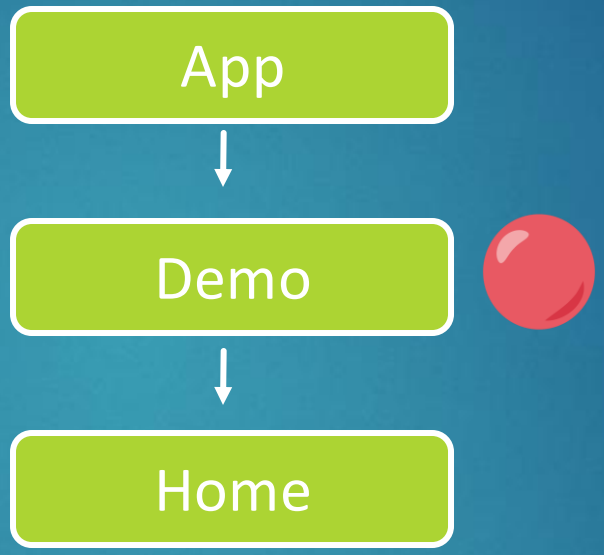
```
// ChildComponent.js
import React from 'react';

function ChildComponent(props) {
  return (
    <div>
      <p>{props.message}</p>
    </div>
  );
}
```

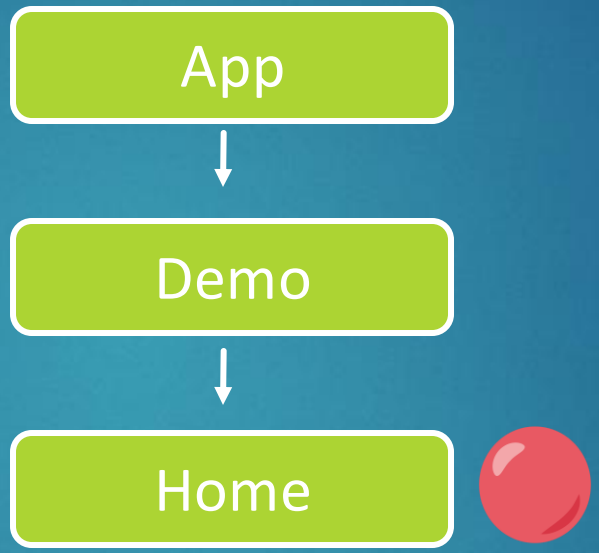
# Props



# Props

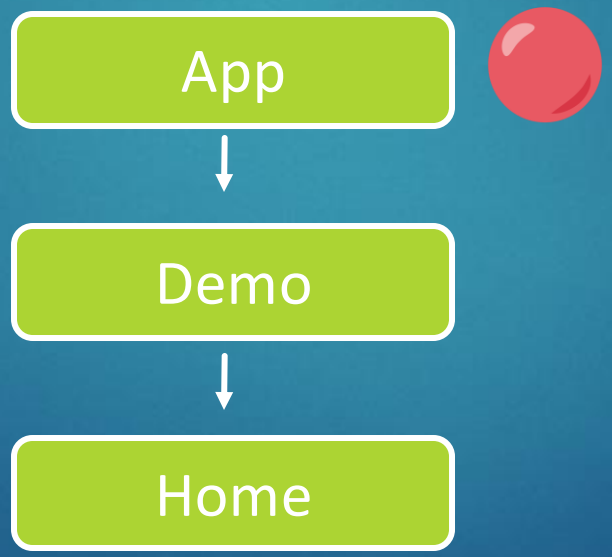


# Props



# One Way Data Flow

In React.js, data primarily flows from parent to child components, following a one-way data flow model. This means that parent components can pass data (via props) down to their child components, but child components cannot directly modify the props they receive from their parents. This one-way flow of data ensures a predictable and easier-to-maintain architecture.



# Try Your Self

**Assignment: Create a Simple User Card**
**Objective:** Build a basic user card component using React.
**Requirements:**
Create a React application with two components: **App** and **UserCard**.
The **UserCard** component should accept the following user information as props:

- **name** (string): The user's name.
- **email** (string): The user's email address.

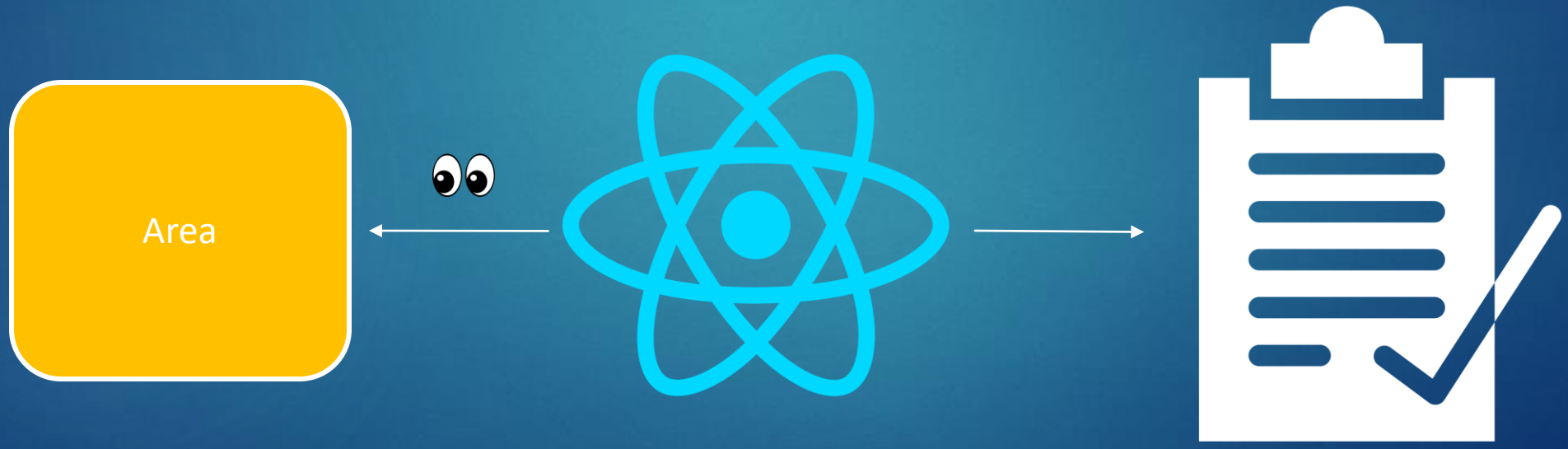
• Inside the **UserCard** component, render the user's name, email.
• In the **App** component, create instances of the **UserCard** component and pass different user information as props to each instance.

# Naming Components

Use **PascalCase**: Component names should always be in PascalCase. This means that the first letter of every word in the component's name should be capitalized. For example: MyComponent, UserProfile, Header.

# React State

- In React, "state" refers to a JavaScript object that holds information about the component.
- It represents the data that the component will **render** and can change over time due to user interactions, network requests, or other factors.



# React Hooks

React Hooks are functions that allow you to use state and other React features in functional components.

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

class Example extends Component {
  constructor() {
    super();
    this.state = {
      count: 0
    };
  }

  incrementCount = () => {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}

export default Example;
```

Class Based

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

function Example() {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}

export default Example;
```

Function Based

# Types of React Hooks

- useState
- useEffect
- useContext
- useReducer
- useRef

# useState

useState is a React Hook that allows functional components to have local component state. It is used to add stateful behavior to functional components, which were previously stateless.

```
function Example() {
  let count=0;
  function setCount(){
    count++;
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount}>Increment</button>
    </div>
  );
}
```

```
import React, { useState } from 'react';

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

  const handleClick = () => {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>
        Click Me
      </button>
    </div>
  );
}

export default Button;
```

# useState

The useState hook returns an array with two elements:

- The current state value.
- A function that allows you to update the state.

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

It is the value that you want to initialize the state with. It can be of any data type (string, number, boolean, object, etc.).

This is the current value of the state.

This is a function that you can call to update the state.
Example - When you call **setState**, it will trigger a re-render of the component with the new state value.

# React Hooks Rules

**Only Call Hooks at the Top Level**

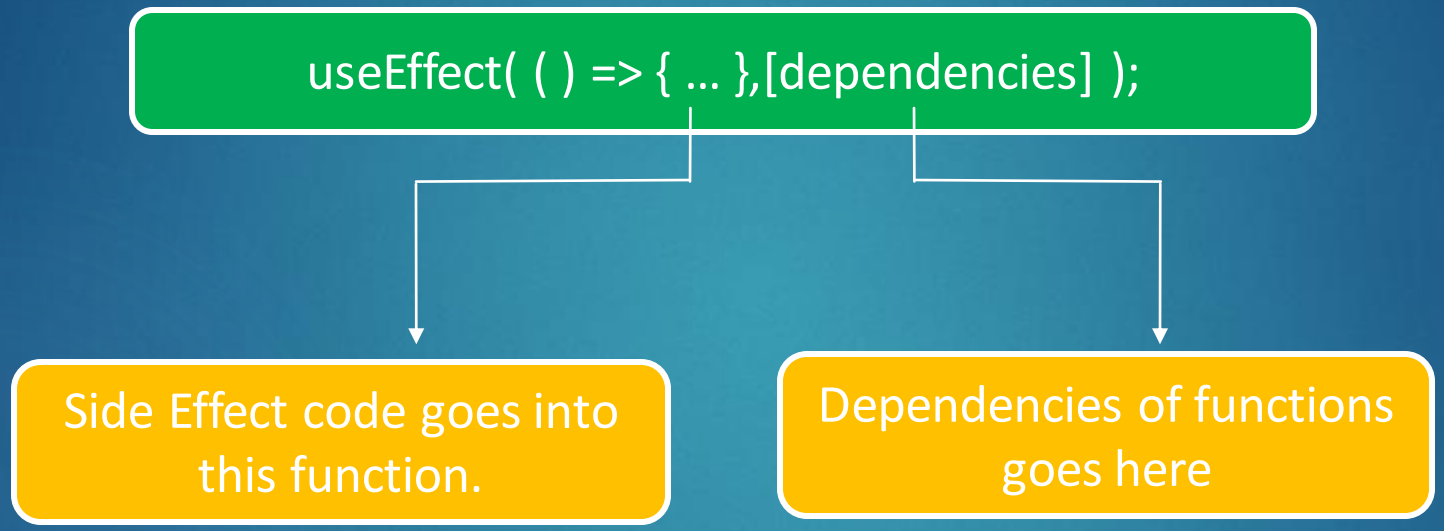
Hooks should be called in the top-level of your functional component or a custom Hook. Don't call Hooks inside loops, conditions, or nested functions.

# Dealing With Side Effects

# useEffect Hook

- useEffect is a hook provided by React, a popular JavaScript library for building user interfaces. It is used to perform side effects in function components.
- Side effects in React components refer to operations that affect the external world, such as making API requests, storing data in Browser Storage, sending http requests to backend servers, and so on.
- The **useEffect** hook allows you to perform these side effects in a functional component in a way that is both efficient and safe.

# useEffect Hook



# useEffect Hook

| | | |
|---|---|---|
| useEffect( ( ) => { … }); | Called after first render | Called after every render |
| useEffect( ( ) => { … },[ ]); | Called after first render | Never called again |
| useEffect( ( ) => { … },[ dependencies ]); | Called after first render | Called after renders if dependencies changes |

# Cleanup function

- In React, the cleanup function in the useEffect hook is a mechanism to perform any necessary clean-up operations before the component is unmounted or before the effect runs again due to changes in dependencies.

- The cleanup function in React's **useEffect** is like a janitor. It makes sure that everything is tidy before a component disappears from the screen or before it gets a makeover because something it depends on changed. It helps prevent messy situations and keeps things running smoothly.