

TECHNOLOGY



ReactJS

React Introduction and Basic Concepts



Learning Objectives

By the end of this lesson, you will be able to:

- 🕒 Create the default project structure of a React.js project
- 🕒 List the components in the ReactJS architecture
- 🕒 Identify the steps in the event handling process



A Day in the Life of a ReactJS Developer

Sarah is a passionate developer who loves creating beautiful and functional websites. Recently, she discovered the power and popularity of React.js, a JavaScript library for building user interfaces.

To work with React.js, she needs to learn the steps to install React.js, how the folder structure is organized, about component-based architecture and the components, as well as event handling, event binding, and methods.

By mastering these key concepts, Sarah will be able to successfully begin her journey as a React.js developer.

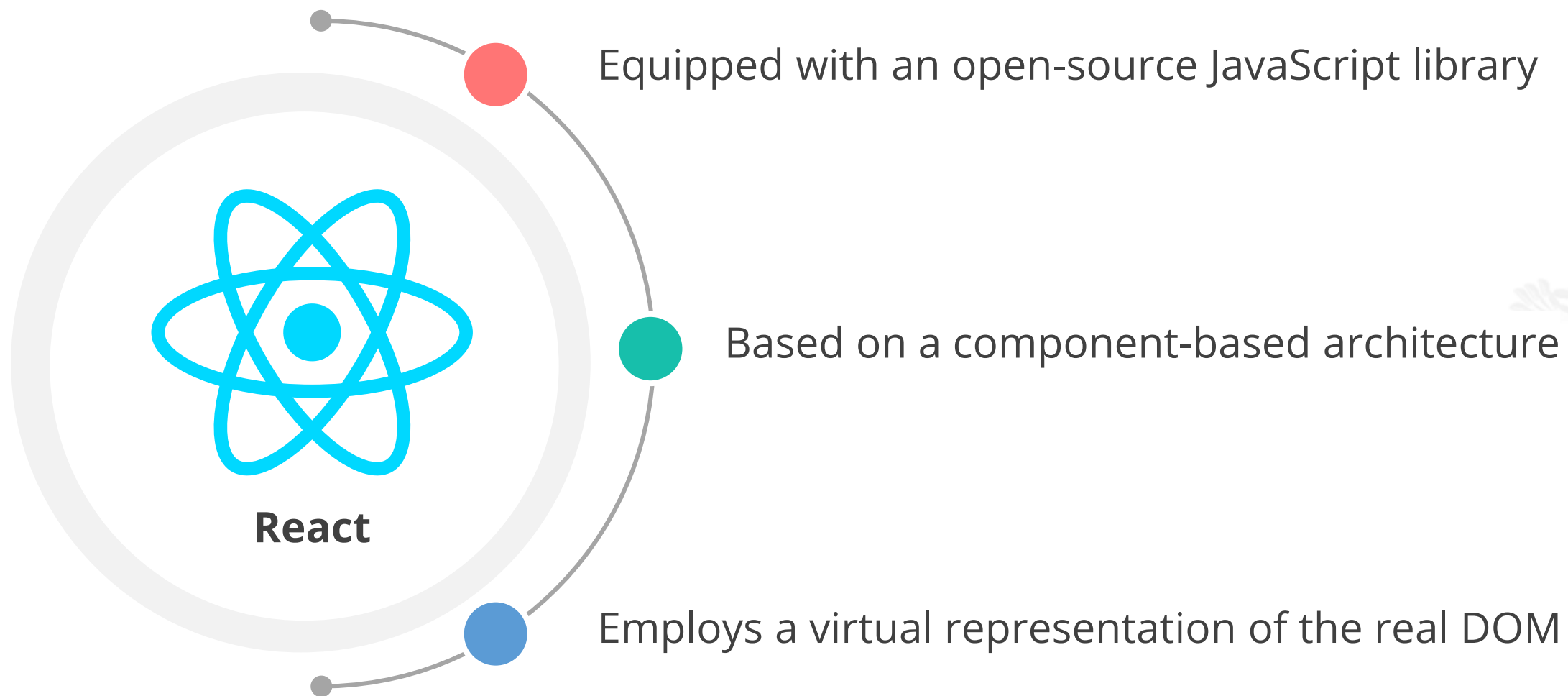


TECHNOLOGY

React

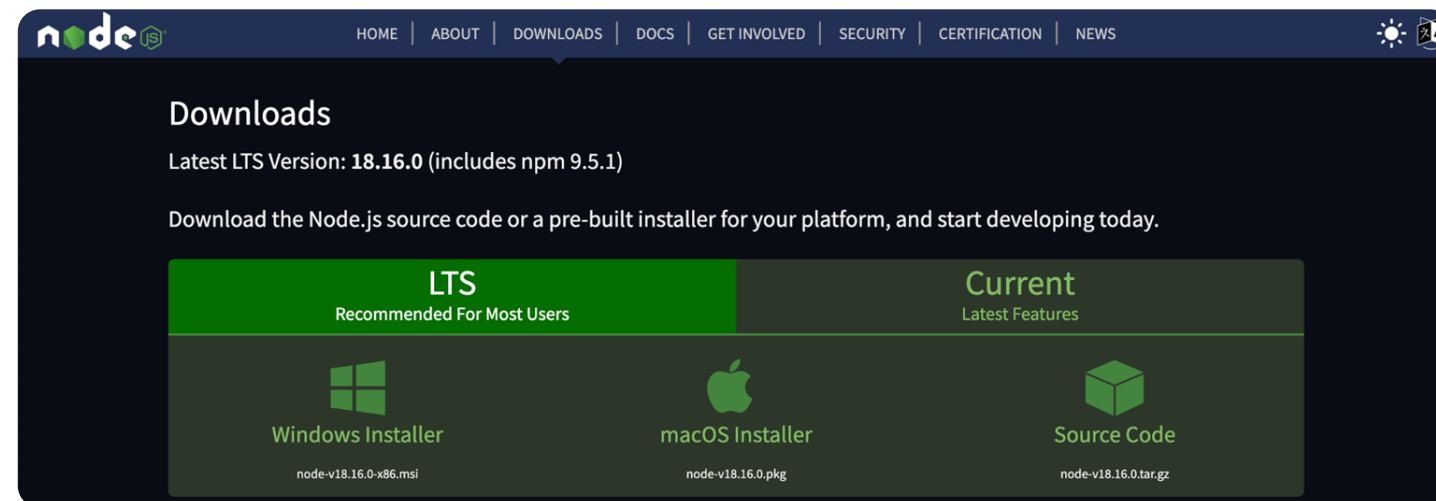
React: An Overview

React is a JavaScript-enabled, open-source library that is used for creating frontend of an application.



Installation of React

Getting started with React requires installing the necessary tools and dependencies to create and run React applications.



Steps to be followed:

1. Go-to node.js download page:
<https://nodejs.org/en/download/>
2. Download the LTS version of the operating system
3. Install the downloaded file

Installation of React

To verify the installation, open the command prompt and check **node -v** and **npm -v** to ensure the packages are downloaded correctly.

```
# Open the command prompt or terminal
# Run the following commands

# Check Node.js version
node -v

# Check npm version
npm -v
```



Installation of React

To create a new React project, install the create-react-app command-line tool globally on the system.

```
npm install -g create-react-app
```



Start a New Project

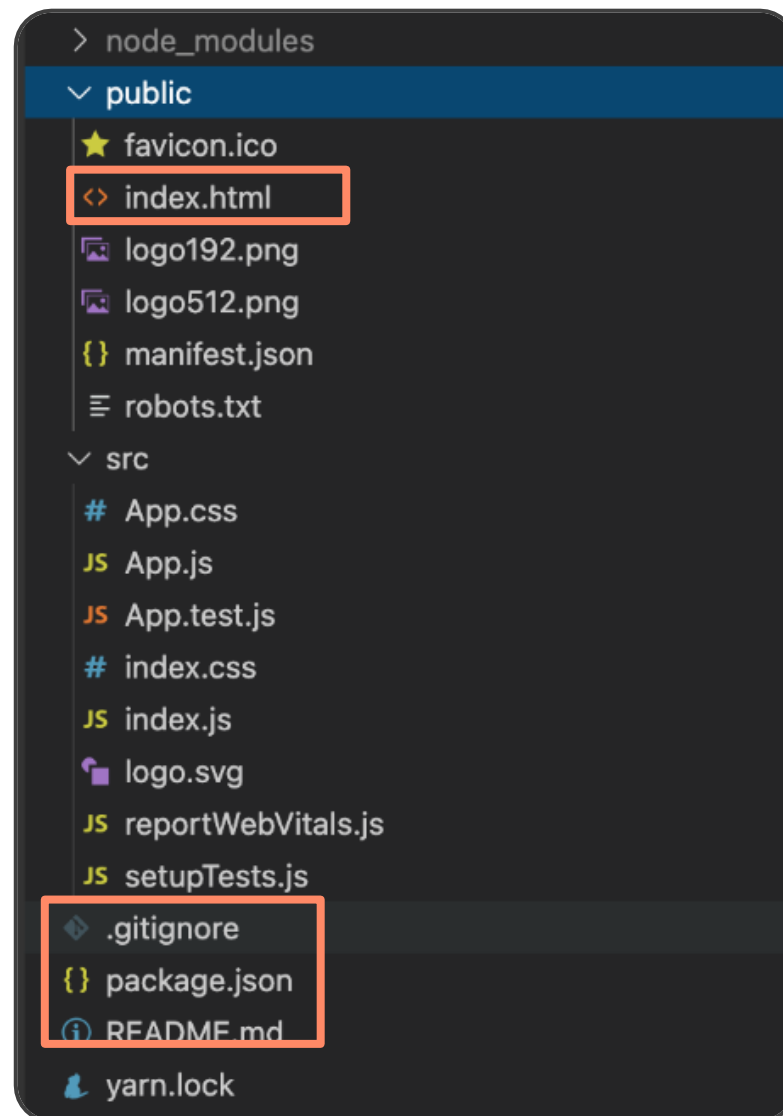
Navigate to the directory where the React project needs to be created and start downloading the development server to begin a new project.

```
cd my-react-app npm start
```



Folder Structure

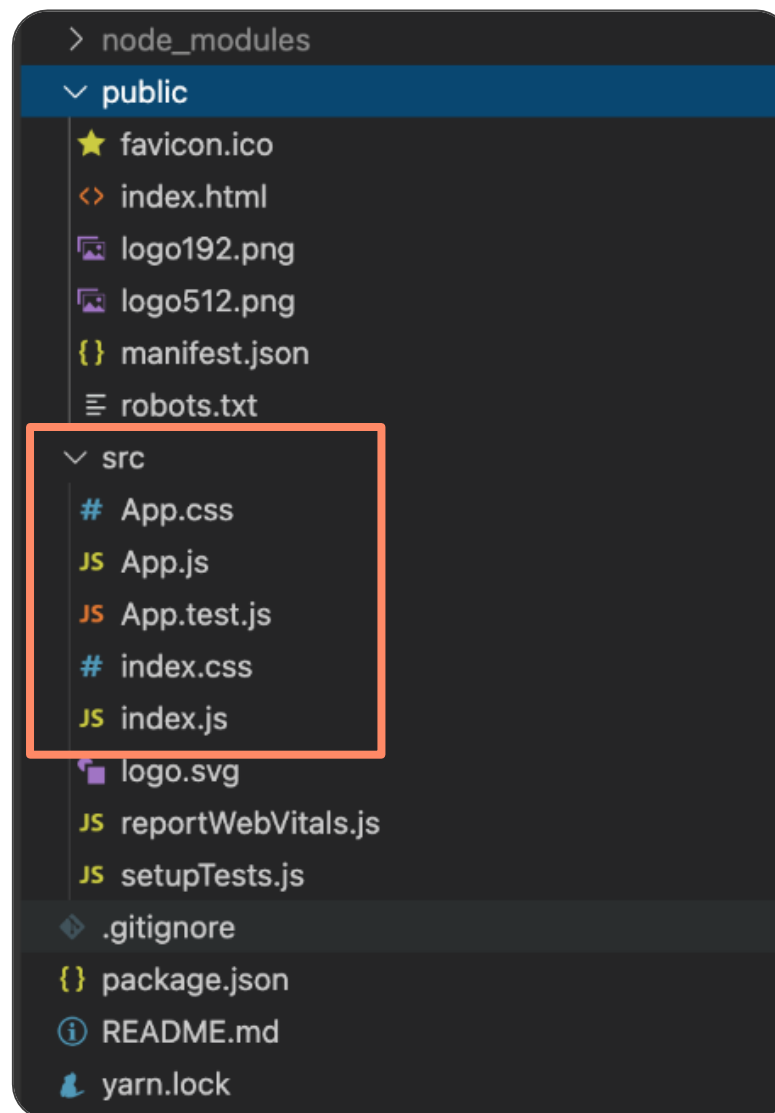
The default project structure consists of several directories, such as public, src, and the node_modules folder.



- In **/public**, the file that is necessary is index.html, which is equivalent to the static index.html file
- **.gitignore** is used for adding the files that git will ignore
- **package.json** includes all the package versions
- **README.md** is the marked-down file used for adding the summary at the bottom of the project

Folder Structure

The /src directory is important as it contains all the code related to a React application.



- The **/src** directory includes all the React-related codes.
- **/src/App.js** is the primary component of an application. Often, it is referred to as the App component.
- **/src/App.css** allows the developer to define the visual appearance and layout of components and elements.
- **/src/index.js** has been assigned the task of rendering the root component into the DOM.

Programming in React: Prerequisites

Before starting a program, it is important to:

Delete all the files from the /src directory and only keep index.js

Import ReactDOM, React, and append a component of React



Programming in React: An Example

A simple code in React is illustrated as follows:

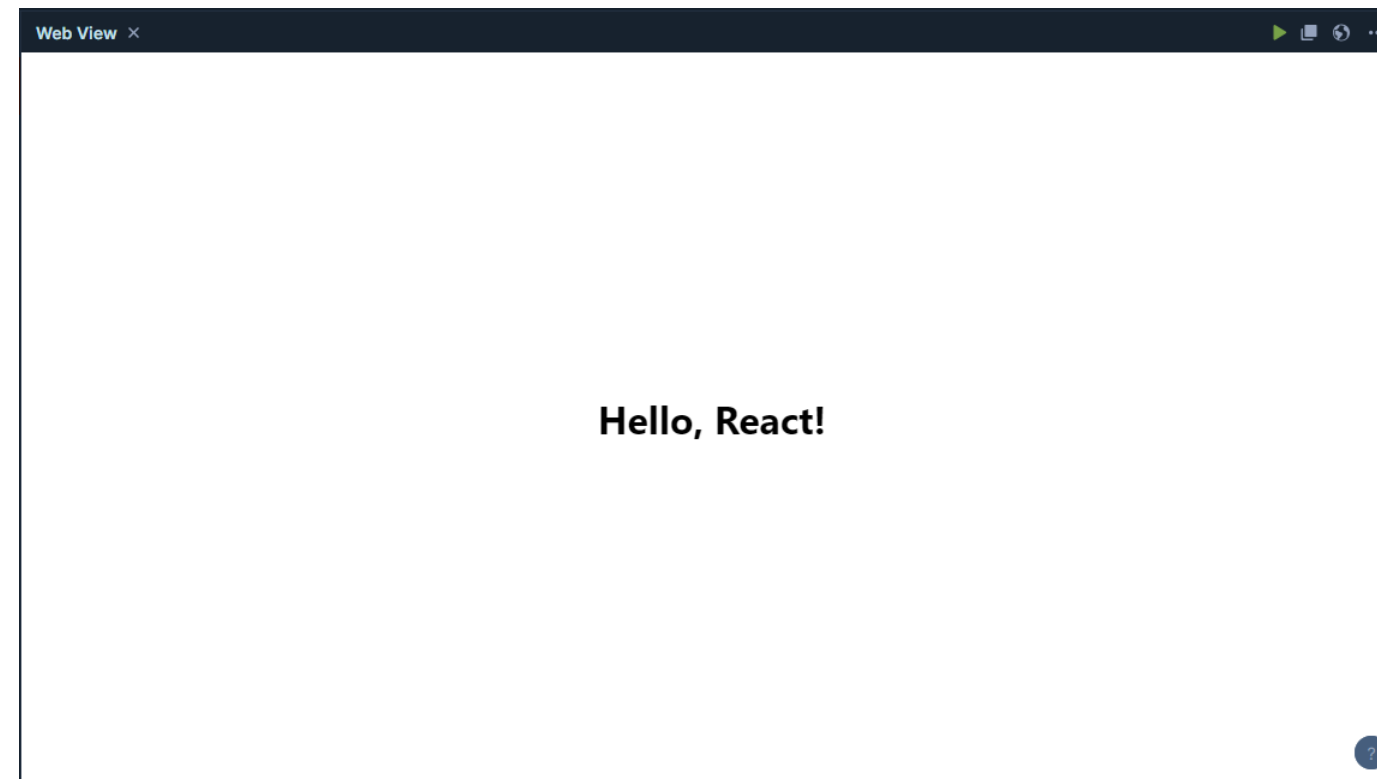
```
import React from "react";
import ReactDOM from "react-dom";
const App = () => {
  return (
    <div>
      <h1>Hello, React!</h1>
    </div>
  );
};
ReactDOM.render(<App />,
  document.getElementById("root")); \
```



Programming in React: An Example

The returned output of the App function is JSX (an extension to JS), which allows writing functions similar to HTML-like syntax.

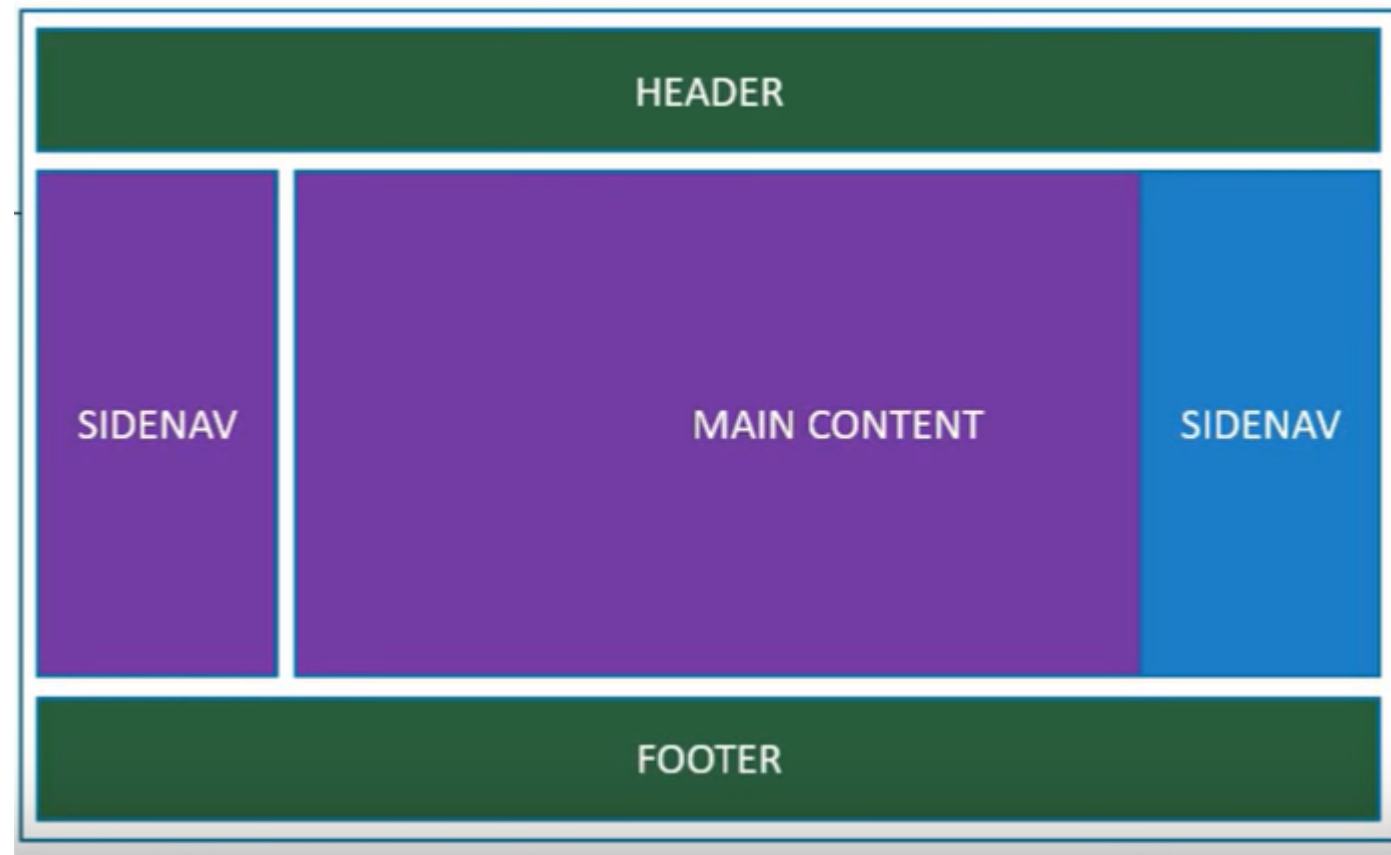
After saving the code and checking the browser, the output will appear as given below:



Screenshot from a web compiler

React Components: An Overview

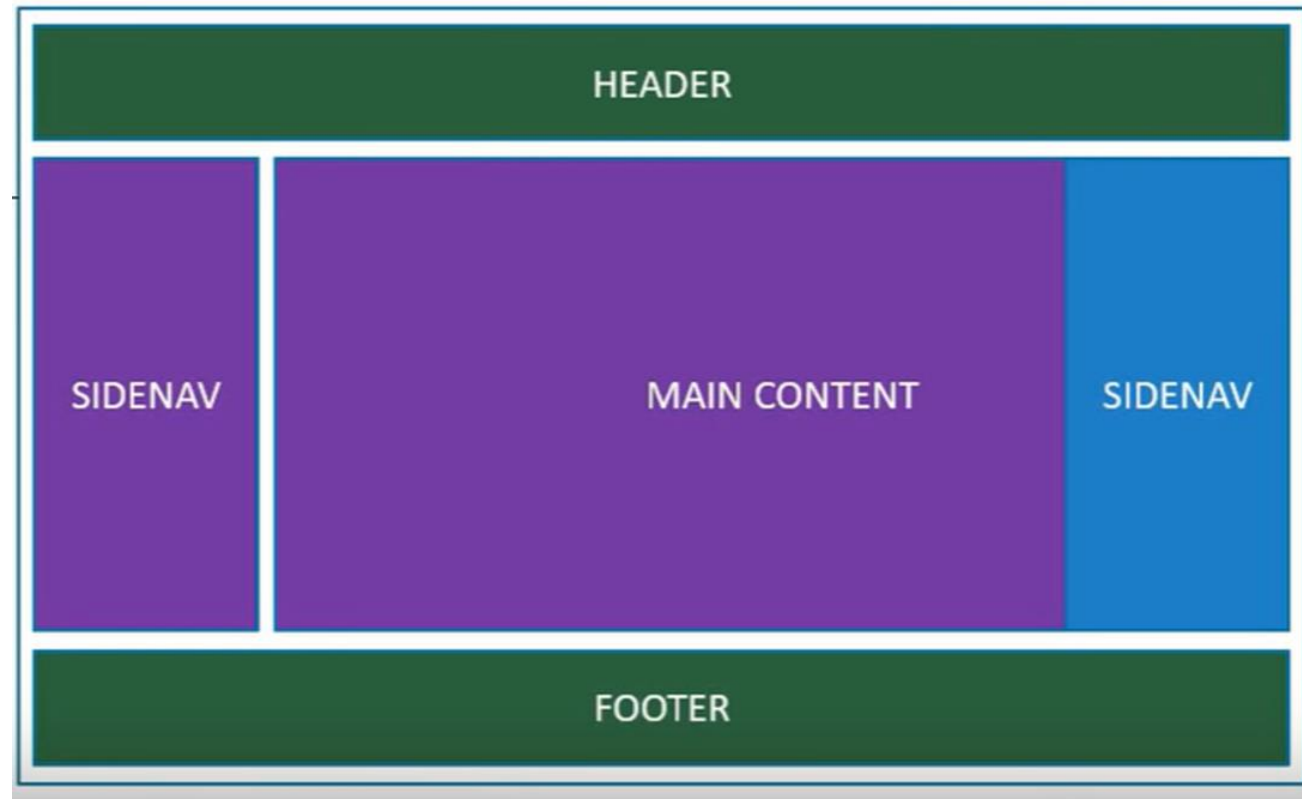
As React is a component-based architecture, developers can divide the application into individual units.



Each component is responsible for:

- Rendering a specific part of the UI
- Handling its own logic
- Creating a hierarchy

React Components: An Overview

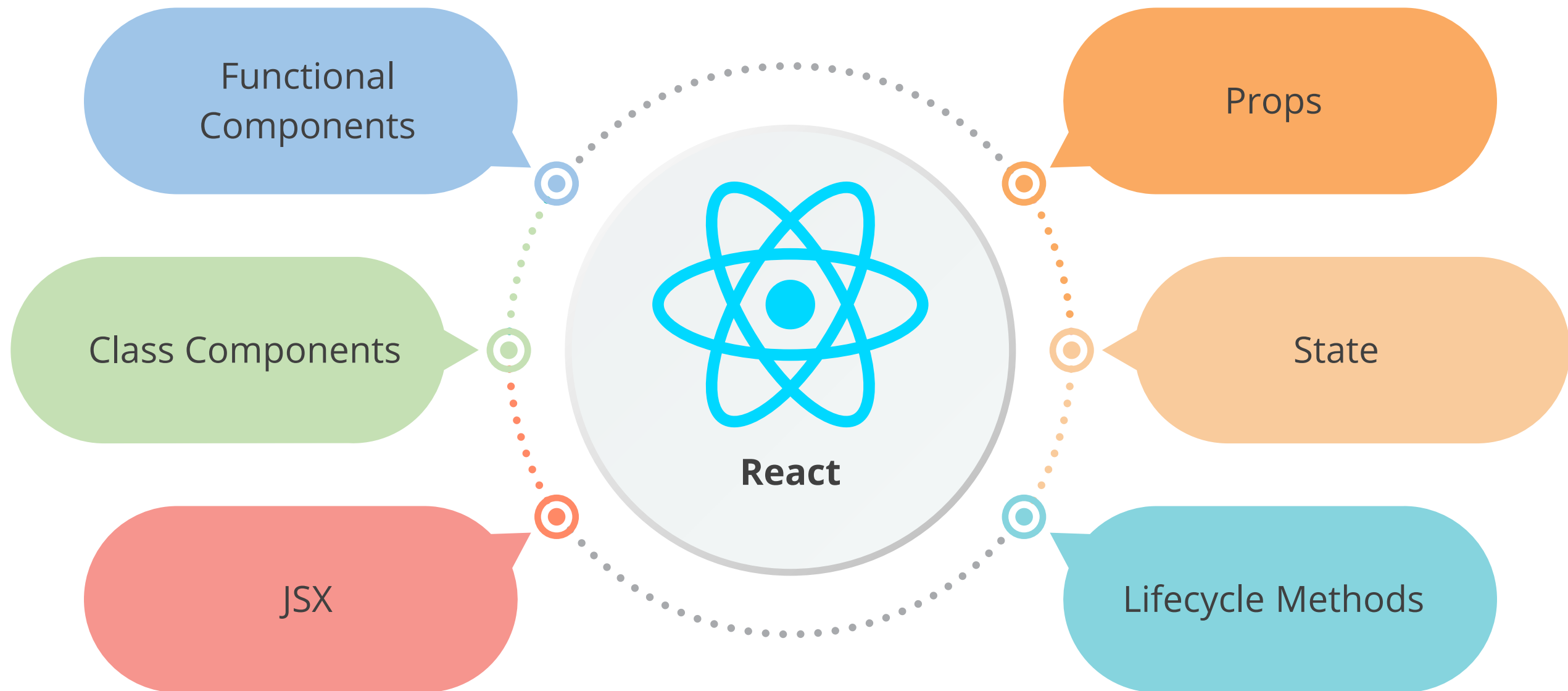


- Each component constitutes a part of the user interface
- It resembles a building block in React
- It is reusable

Example: The side navigation component can be placed either on the left side or the right side.


Components in React

Here are the following types of components in React:



Functional Components

Functional components can be defined and developed using JavaScript functions. They are popular because of the following features:



Simplicity
and
Readability

Reusability

Good
Performance

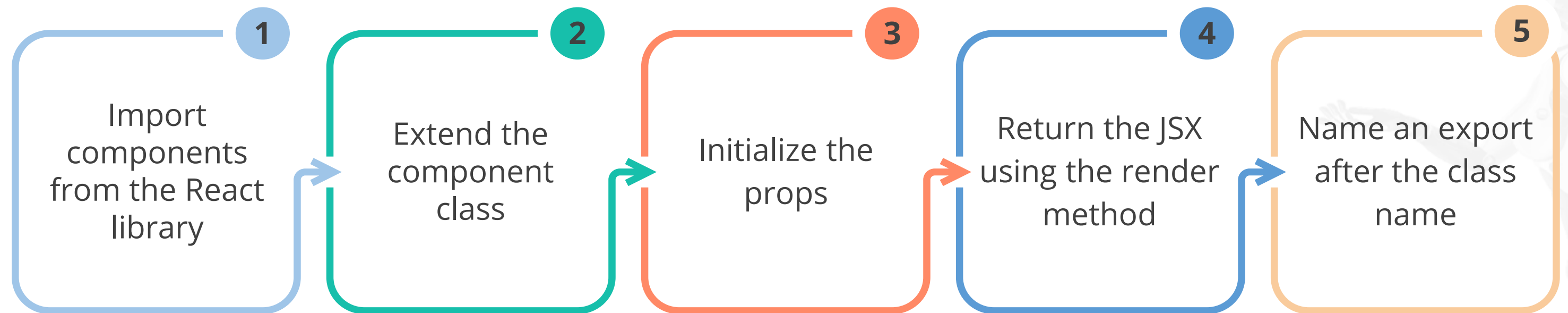
Hooks

Testing

Class Components

A class component defines and creates reusable UI components using JavaScript classes.

To create a class component:



Class Components: An Example

The following example shows how a class component can be developed.

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello,  
    {this.props.name} </h1>;  
  }  
}
```

Example:

Class **Welcome** extends **React.Component**. A render method of the class returns a <h1> tag (**Hello**).

Functional and Class Components

Developers have a preference for functional components instead of class components. Here are some of the reasons why they do so:

Simplicity

Functions accept props and return a declaration

No this Keyword

this keyword is absent, which makes the task easier for beginners

No State

A state with many components is not needed to find a solution

Complex Logic

Logic is not complicated

Demo with React Components



Duration: 30 min.

Problem Statement:

You are given a project to create and start a React application with a class component displaying a text on the screen.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a class component
2. Implement a render function to return the h2 element




Hooks: An Overview

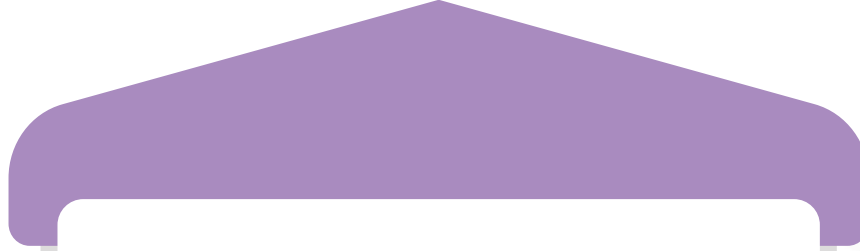
Hooks allow the addition of custom behavior to an application, enabling more flexibility. Some of the important features of Hooks are as follows:



Follow specific rules and naming conventions



Called only at the top level of a functional component



Use the prefix **use**, followed by the feature they provide

Hooks: An Overview

Hooks enable coders to use functional components in coding without the help of class syntax. They revolutionized the method of writing React components by allowing users to:



Use the features of components in functional components

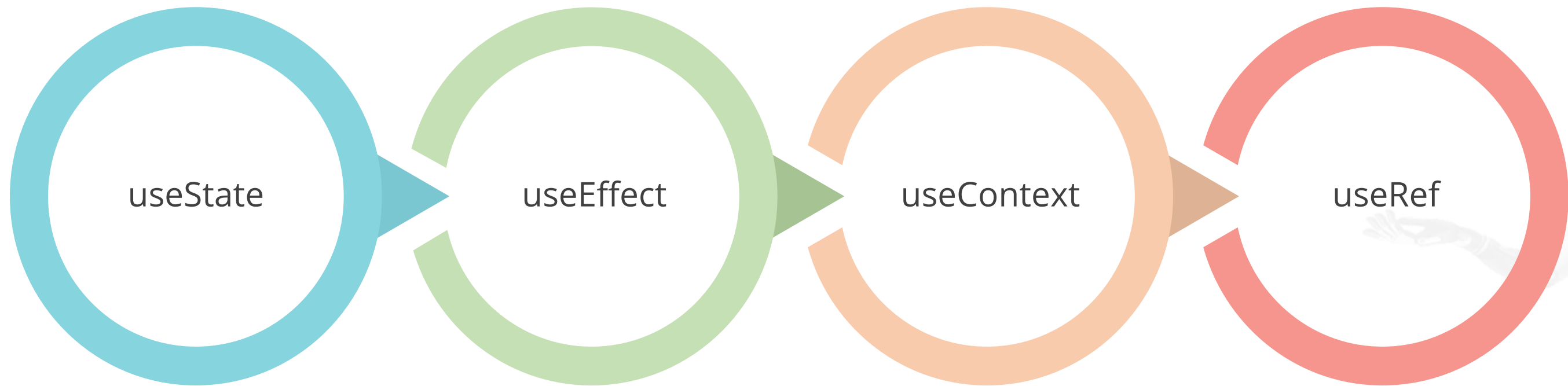
Add features for state and lifecycle

Arrange the component logic into units that can be used



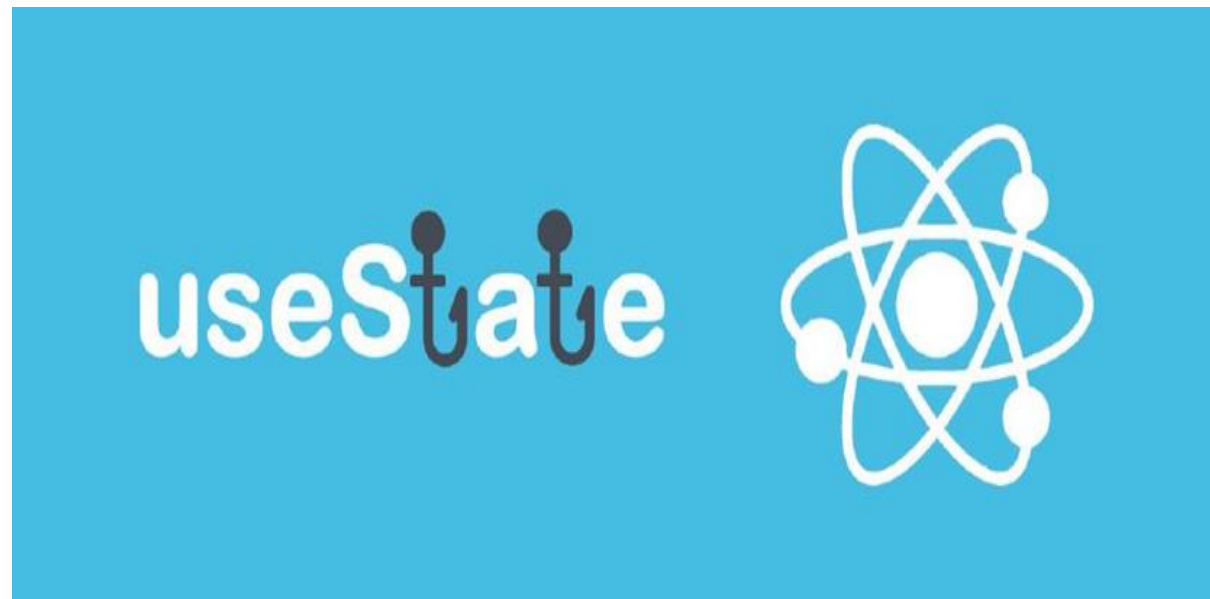
Types of Hooks

Here are the following Hooks that help to manage state, handle side effects, share data between components, and more:



The useState Hook

The **useState** Hook is vital as it permits functional components to have their own state.

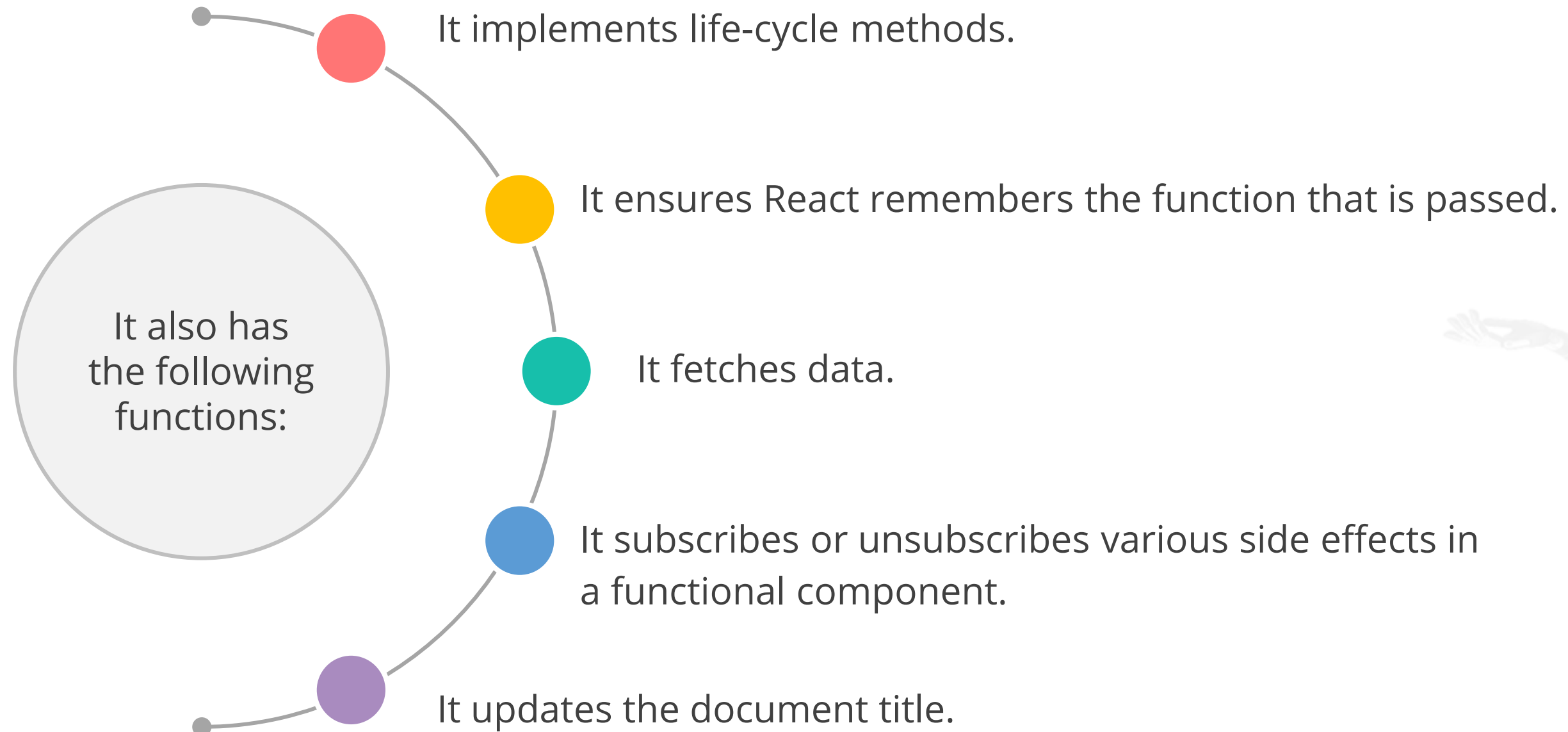


Source: <https://jeffdevslife.com/p/useState-hook/>

- **useState** can be used with an array, object, string, boolean, or number.
- The **useState** function picks the initial state as the argument.
- **useState** Hooks return a pair of values as an array.

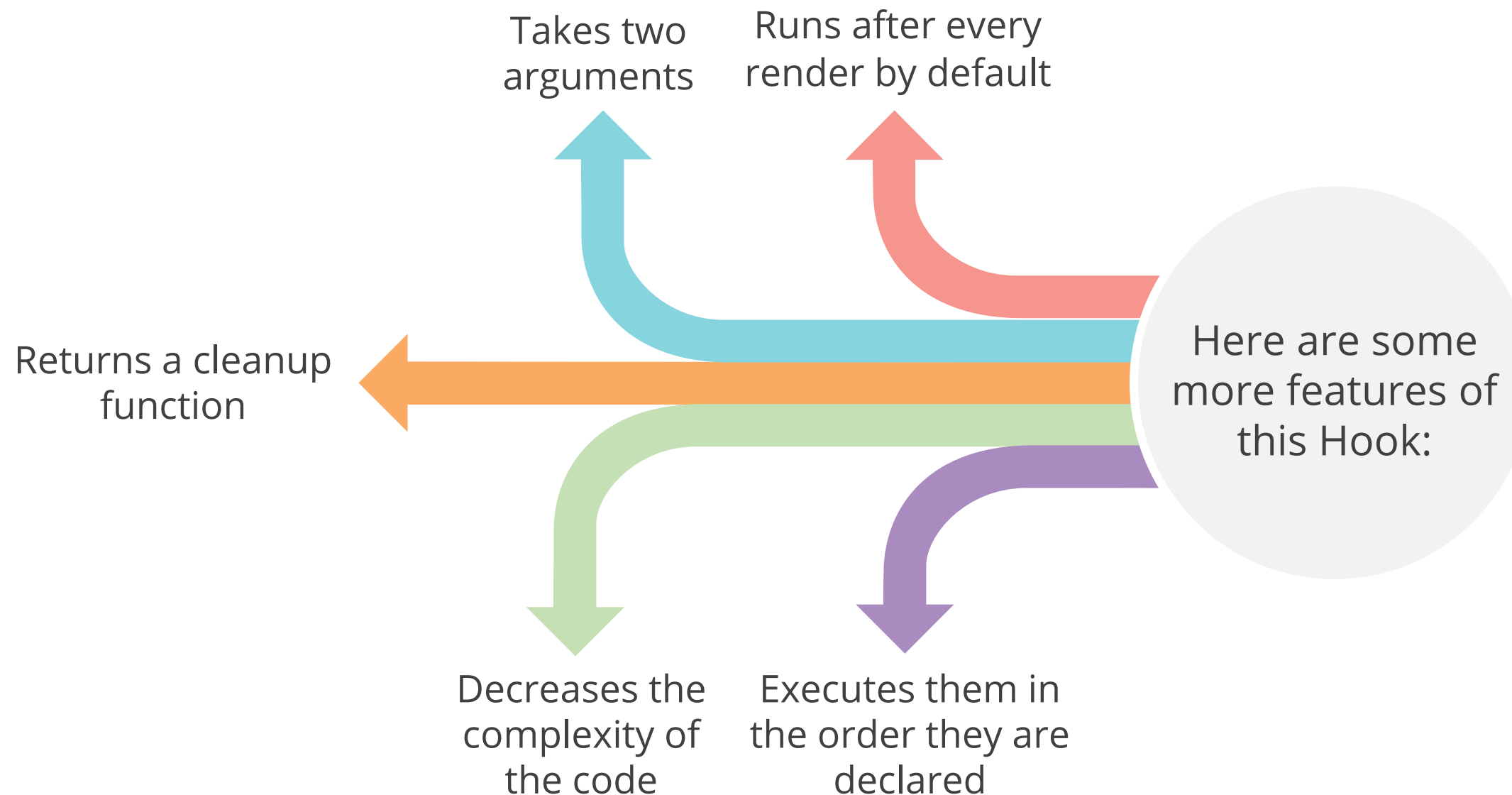
The `useEffect` Hook: Functions

The **`useEffect`** Hook ensures side effects are performed in functional components.



The `useEffect` Hook: Features

The **`useEffect`** Hook controls the execution of the side effects and keeps the component's behavior consistent.



JSX Syntax

JSX (JavaScript XML), a syntax extension, ensures coders can write code similar to HTML in JavaScript files. Some of the important features of JSK are as follows:

It is an extension of JavaScript.

It builds user interfaces.

It combines JavaScript and HTML into one language.

It changes into regular JavaScript during runtime.

It embeds JavaScript logic.

Benefits of Using JSX

Developers use the JSX component in React applications for the following reasons:

- Allows HTML-like code in JavaScript files
- Efficiently creates interactive UI components
- Allows declarative and component-based code
- Allows dynamic content and conditional rendering
- Improves code organization and reusability
- Provides intuitive syntax for defining UI elements



Props in React

This code shows how props can be passed as attributes to a component. It also displays how to use the **props** parameter.

```
// Greet.js
import React from 'react';
const Greet = (props) => {
  console.log(props);
  return <h1>Hello, {props.name}!</h1>;
};
export default Greet
```



Props in React

The code shows how props can render dynamic content.

```
// App.js
import React from 'react';
import Greet from './Greet';
const App = () => {
  return (
    <div>
      <Greet name="John" />
      <Greet name="Mary" />
      <Greet name="Rock" />
    </div>
  );
};
export default App;
```



Using Props in React

The following steps are required:

- To display the value of a specific prop, use the syntax `props.name`, where `name` is the attribute name
- In JSX, to include the value of a prop inside the rendered output, wrap the prop expression in curly braces `{}`

```
4 function List() {  
5   const students = [  
6     {  
7       name: 'Adil',  
8       id: 1  
9     },  
10    {  
11      name: 'Arman',  
12      id: 2  
13    },  
14    {  
15      name: 'Ahmet',  
16      id: 3  
17    },  
18    {  
19      name: 'Anwar',  
20      id: 4  
21    }  
22  ];  
23  
24  return <NewList />;  
25  // const names = students.map((student) => <li key={student.id}>{student.name}</li>  
26  // return <ul style={{ color: 'red' }}>{names}</ul>;  
27 }  
28  
29 export default List;
```

State in React

JavaScript object **state** is used to hold data for a component.

For instance, a state object can be initialized inside a constructor.

```
class Truck extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {brand: "Ford"};  
  }  
  render() {  
    return (  
      <div>  
        <h1>My Truck</h1>  
      </div>  
    );  
  }  
}
```



Using State in React

A state object can be used with the syntax: **this.state.propertyname**, anywhere in a component.

```
class Truck extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      brand: "Ford",  
      model: "Pickup",  
      color: "red",  
      year: 1994  
    };  
  }  
  render() {
```



Using State in React

The code represents a component that renders information about a specific product, including its brand, color, model, and year, using JSX syntax in React.

```
return (  
  <div>  
    <h1>My {this.state.brand}</h1>  
    <p>  
      This is {this.state.color}
```

```
      {this.state.model}  
      from {this.state.year}.  
    </p>  
  </div>  
  );  
}
```

setState in React

The component class provides a method, **setState**, that updates a component's state.



When a developer calls setState, React:

- Merges the provided object with the current state object
- Updates the specified properties

Source: <https://www.poudelanil.com.np/react-setstate-callback/>

setState in React: Example

Example: Change the color of the truck from red to blue with a click of a mouse button. For this, use the **onClick** event, which is called when the mouse button is clicked.

```
class Truck extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Pickup",
      color: "red",
      year: 1964
    };
  }
  color_change = () => {
    this.setState({color: "blue"});
  }
}
```



setState in React: Example

Example: Here, the **render() method** of the Truck component returns a JSX structure that represents the visual output of the component.

```
render() {  
  return (  
    <div>  
      <h1>My {this.state.brand}</h1>  
      <p>  
        It is a  
{this.state.color}  
        {this.state.model}  
        from {this.state.year}
```

```
</p>  
      <button  
        type="button"  
        onClick={this.changeColor}  
      >Change color</button>  
    </div>  
  );  
}
```

Destructure Props and State

In React, destructuring props in the state refers to extracting specific properties from the props object.

Consider an employee object with the given properties:

```
const employee = {  
  first_name: "Ramesh",  
  last_name: "Fadatare",  
  email_Id:  
  "ramesh@gmail.com"  
}
```

Before ES6, each property had to be accessed individually using dot or bracket notation.

```
console.log(employee.first_name) // Ramesh  
console.log(employee.last_name) // Fadatare  
console.log(employee.email_Id) // ramesh@gmail.com
```

Destructure Props and State

Destructuring of Props and State properties can efficiently extract and assign values in a concise manner. The following code illustrates this point:

Destructuring helps in streamlining the given code:

```
const { first_name,  
last_name, email_Id } =  
employee;
```

The code is similar to the following code:

```
const first_name =  
employee.first_name  
const last_name =  
employee.last_name  
const email_Id=  
employee.email_Id
```

So now, these properties can be received:

```
console.log(first_name)  
// Ramesh  
console.log(last_name)  
// Fadatare  
console.log(email_Id) //  
ramesh@gmail.com
```

Destructure Props and State: An Example

The given code is a functional component called Employee in React that displays employee details.

```
import React from 'react'
export const Employee = props => {
  return (
    <div>
      <h1> Employee Details</h1>
      <h2> First Name :
{props.first_name} </h2>
```

```
<h2> Last Name : {props.last_name} </h2>
      <h2> Email Id : {props.email_Id}
</h2>
    </div>
  )
}
```

Note that the employee data can be accessed in the Employee component using props.

Destructure Props and State: An Example

This code represents the main entry point of a React application. The Employee component receives props for **firstName**, **lastName**, and **emailId**.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Table from
'./components/functional-
components/Table';
import PropsDemo from
'./components/props/PropsDemo';
import { Employee } from
'./components/Employee';
function App() {
```

```
  return (
    <div className="App">
      <header className="App-header">
        <Employee first_name =
        "Ramesh" last_name = "Fadatare"
        email_Id = "ramesh@gmail.com" />
      </header>
    </div>
  );
}
export default App;
```

Destructure Props: Function Parameter Destructured

Here is one way to destructure props in a functional component:

1. Function parameter destructured

```
import React from 'react'
export const Employee = ({first_name, last_name, email_Id})
=> {
  return (
    <div>
      <h1> Employee Details</h1>
      <h2> First Name : {first_name} </h2>
      <h2> Last Name : {last_name} </h2>
      <h2> Email Id : {email_Id} </h2>
    </div>
  )
}
```

Destructure Props: Props Destructured

2. Props destructured in the function body

```
import React from 'react'
export const Employee = props => {
  const {first_name, last_name, email_Id} = props;
  return (
    <div>
      <h1> Employee Details</h1>
      <h2> First Name : {first_name} </h2>
      <h2> Last Name : {last_name} </h2>
      <h2> Email Id : {email_Id} </h2>
    </div>
  )
}
```



Destructure Props Using Class Components

Specific props are picked up from the **this.props** object and allotted to individual variables when destructuring props in class components.

Here is an example of a program without destructuring:

```
import React, { Component } from
'react'
class Employee extends Component {
  render() {
    return (
      <div>
        <h1> Employee
        Details</h1>
```

```
</div>
      <h2> First Name :
      {this.props.first_name} </h2>
      <h2> Last Name :
      {this.props.last_name} </h2>
      <h2> Email Id :
      {this.props.email_Id} </h2>
    </div>
  )
}
}
export default Employee;
```

Destructure Props Using Class Components

In class components, destructure props in the **render() function** are shown here:

```
render() function:
import React, { Component } from
'react'
class Employee extends Component {
  render() {
    const {first_name, last_name,
email_Id} = this.props;
    return (
      <div>
        <h1>
```

```
Employee Details</h1>
          <h2> First Name :
{first_name} </h2>
          <h2> Last Name :
{last_name} </h2>
          <h2> Email Id :
{email_Id} </h2>
        </div>
      )
    }
  }
  export default Employee;
```

Destructure State

This code is an example of a class component in React that renders the details of an employee.

```
import React, { Component } from
'react'
class StateDemp extends Component {
  render() {
    const {state_1, state_2,
state_3} = this.state;
    return (
<div>
      <h1> State Details</h1>
      <h2> state1 :
```

```
{state_1} </h2>
      <h2> state2 : {state_2}
</h2>
      <h2> state3 : {state_3}
</h2>
    </div>
    )
  }
}
export default StateDemo;
```

Demo with React Props and State



Duration: 20 min.

Problem Statement:

You are given a project to implement a React web application to store any details of a bike.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Create a class bike with the make, model, and color
2. Create a function to change the bike color using **setState**
3. Render the bike property on the page

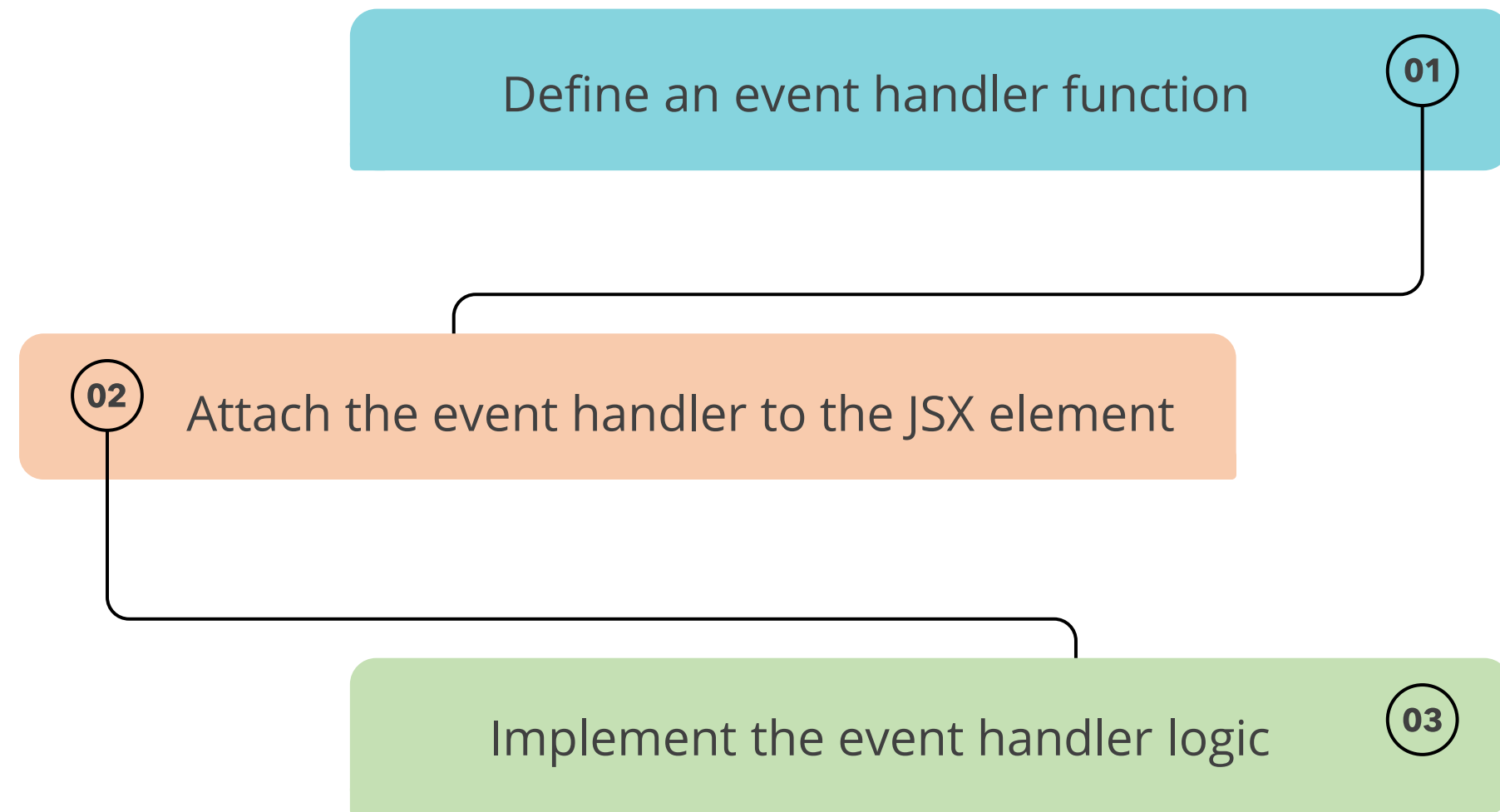


Handling Events

Event Handling vs. DOM

Event handling captures and responds to the actions or events in a component.

The typical steps to handle events in React are:



Event Handling vs. DOM

In React, event handling allows one to respond to user interactions such as button clicks, form submissions, or keyboard input.

Features of React and DOM event handling are as follows:

React events use camelCase names

Example: OnClick

DOM events use lowercase names

Example: Onclick

React events require a function as an event handler

onClick=
{buttonClicked}

DOM events can use a string as an event handler

onclick=
"buttonClicked()"

Event Handling vs. HTML

React event handling and HTML event handling have some similarities, but there are also important differences between them.

HTML

```
<button onclick="buttonClicked()">  
  Button Click!  
</button>
```

Event handlers are defined as attributes on HTML elements

React

```
<button onClick={buttonClicked}>  
  Button Click! </button>
```

Event handlers are typically passed as props to components

Event Handling vs. HTML

To prevent default behavior in React, **return false** cannot be used, and instead, **preventDefault** must be called explicitly.

HTML

```
//HTML
<button onclick="console.log('Button
Clicked.');
```

return false">
 Click Me
</button>

React

```
// React
Function handleClick (e) {
  e.preventDefault ();
  console.log('Button clicked.');
```

```
}
Return (
  <button onClick={handleClick}>
    Click me
  </a>
);
```

Event Handling: Functional Component

The **FunctionClick** component renders a button. When the button is clicked, the **clickHandler** function is called.

```
import React from 'react'
function FunctionClick() {
  function clickHandler() {
    console.log('Button clicked')
  }
}
```

```
return (
  <div>
    <button
      onClick={clickHandler}>Click</button>
    </div>
  )
}
export default FunctionClick;
```

Event Handling: Class Component

The **ClassClick** component is a class-based component in React that renders a button. When the button is clicked, the **clickHandler** method is called, logging a message to the console.

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

class ClassClick extends
Component {

  clickHandler() {

    console.log('Clicked the
button')

  }

  render() {
```

```
    return (

      <div>

        <button
onClick={this.clickHandler}>Clic
k Me</button>

      </div>

    )

  }

}

export default ClassClick
```


Binding Event Handlers

Event Binding permits coders to state a function or method to be implemented when an event occurs.

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
    super()
    this.state = {
      message: 'Hello'
    }
  }
}
```

When using class methods for event handling, the coder must ensure the correct binding of **this** within the method to access the component's instance and its properties.



Binding Event Handlers

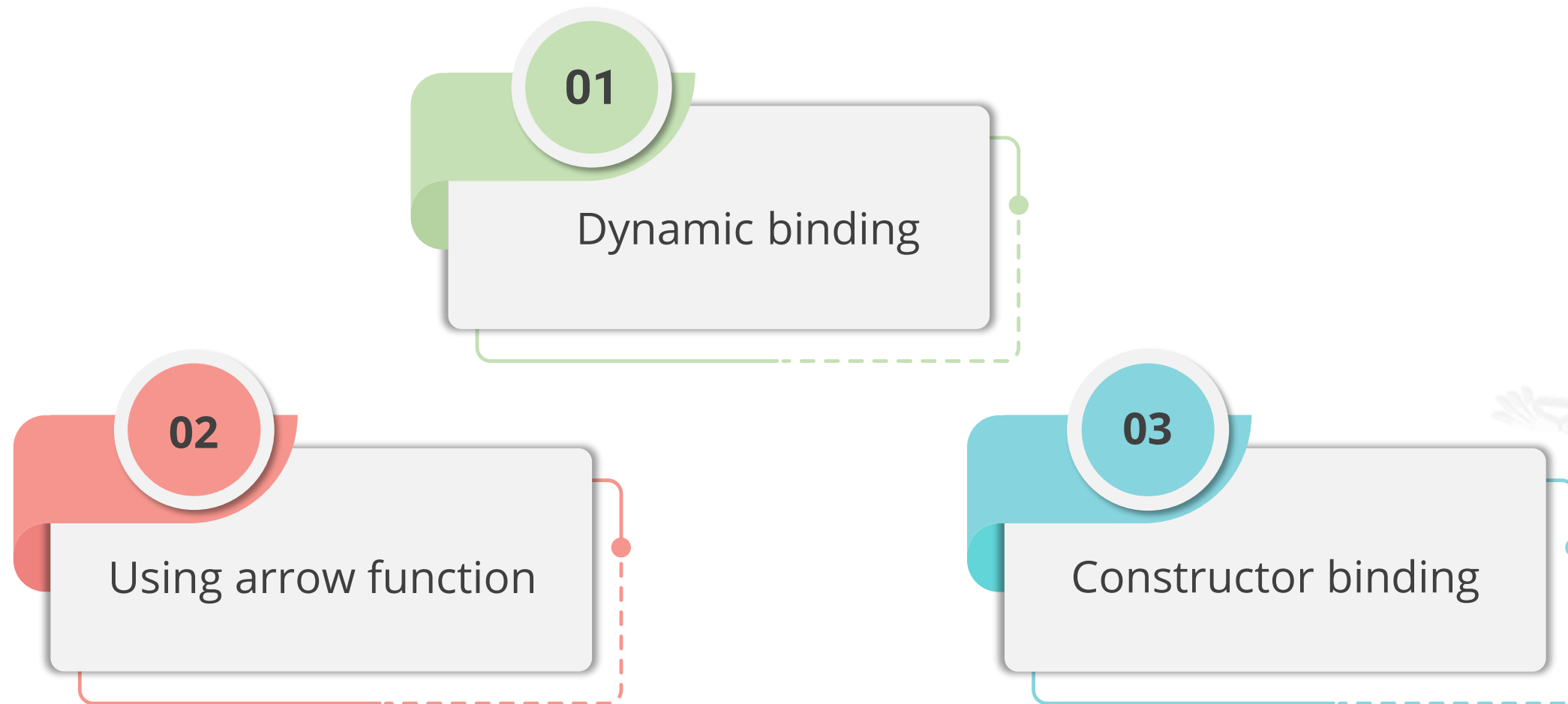
If the binding is not done, the **this** keyword may return undefined, resulting in a **TypeError: Cannot read property setState of undefined error**.

```
clickHandler() {  
  console.log(this)  
  this.setState({message:  
'Goodbye'})  
}  
render() {  
  return (  
    <div>
```

```
<div>{this.state.message}</div>  
    <button  
onClick={this.clickHandler}>Click</b  
utton>  
    </div>  
  )  
}  
}  
  
export default EventBind
```

Binding Event Handlers: Methods

React offers three ways to bind event handlers:



Dynamic Binding

Dynamic binding refers to the process of establishing a connection between a function and its execution context (the value of **this**) at runtime.

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
    super()
    this.state = {
      message: 'Hello'
    }
  }
}
```

It allows for flexible and late binding of functions to objects or instances.



Dynamic Binding

```
clickHandler = () => {  
  this.setState({message: 'Goodbye'})  
}  
render() {  
  return (  
    <div>  
      <div>{this.state.message}</div>  
      <button onClick={this.clickHandler}>Click</button>  
    </div>  
  )  
}
```

Changing **this.state.message** causes the component to re-render and generate a new handler that is different from the one used during the initial call to render().

Using the Arrow Function

With arrow functions, there is no need to explicitly bind the event handler, as they automatically capture the **this** context of the component.

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
    super()
    this.state = {
      message: 'Hello'
    }
    this.clickHandler =
this.clickHandler.bind(this)
  }
  clickHandler() {
    console.log(this)
```

```
    this.setState({message: 'Goodbye'})
  }
  render() {
    return (
      <div>
        <div>{this.state.message}</div>
        <button
onClick={this.clickHandler}>Click</button>
      </div>
    )
  }
}
export default EventBind
```

Constructor Binding

Constructor binding binds the context of a class method to the component.

```
import React, { Component } from 'react'
class EventBind extends Component {
  constructor() {
    super()
    this.state = {
      message: 'Hello'
    }
    this.clickHandler =
this.clickHandler.bind(this)
  }
  clickHandler() {
    console.log(this)
  }
}
```



Constructor Binding

```
this.setState({message: 'Goodbye'})
}
render() {
  return (
    <div>
      <div>{this.state.message}</div>
      <button
onClick={this.clickHandler}>Click</button>
    </div>
  )
}
}
export default EventBind
```

Arrow function and constructor binding provide good performance and are widely used in React applications for binding events.

Demo with Event Handlers



Duration: 15 min.

Problem Statement:

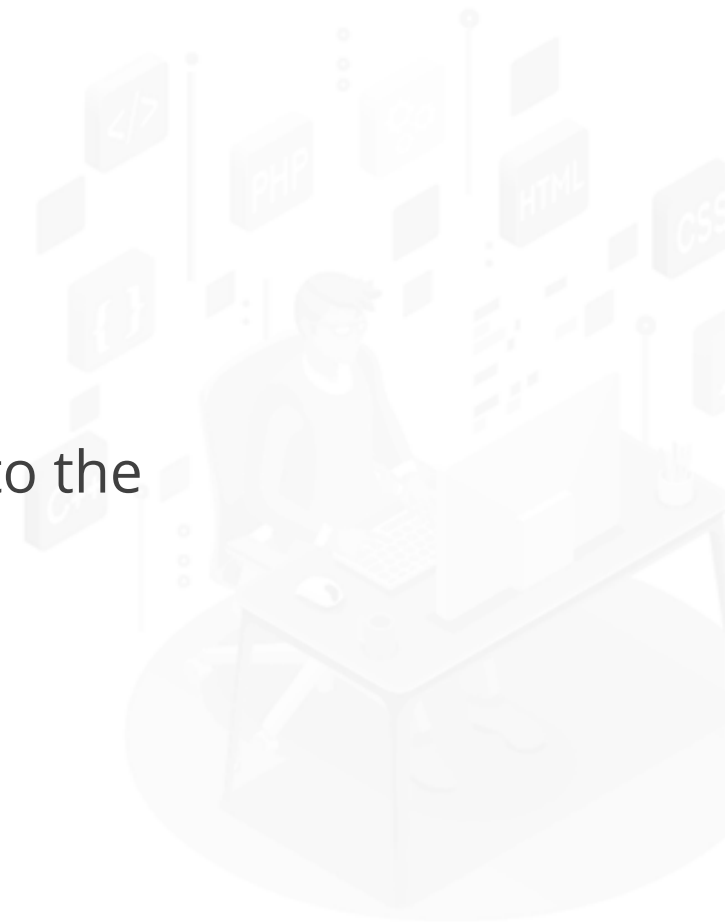
You are given a project to create a React component that binds its event handler contexts and passes data into the event handler.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Import the necessary dependencies from the **React** and **react-dom** libraries
2. Import the **MyList** component from **./MyList**
3. Define an array of items with their IDs and names
4. Render the **MyList** component with the **items** array as a property and mount it to the **root** element in the HTML document
5. Implement the **MyList** component



Methods as Props

Methods as props: a method defined in a parent component is shared with a child component as a prop.

Parent Component

```
import React, { Component } from 'react'
import ChildComponent from '../ChildComponent';
class ParentComponent extends Component {
  constructor(props) {
    super(props)

    this.state = {
      parentName: 'Parent'
    }
    this.greetParent = this.greetParent.bind(this)
  }
}
```



Methods as Props

The child component can then invoke that method when certain events or conditions occur.

```
greetParent() {
  alert('Hello: ' + this.state.parentName);
}
render() {
  return (
    <div>
      { /* We pass a reference to a method as props to the child component
       greetHandler will now be available to the child component */ }
      <ChildComponent greetHandler={ this.greetParent }></ChildComponent>
    </div>
  )
}

export default PerentComponent
```

Methods as Props

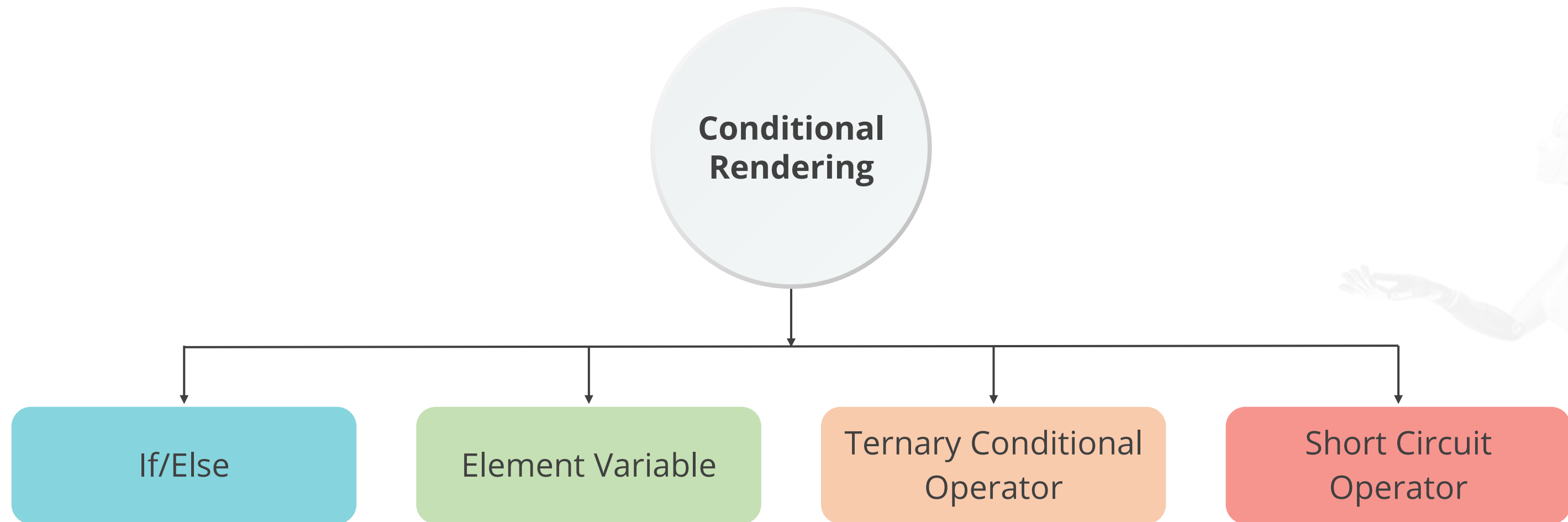
Here, a functional component called **ChildComponent** is used. This component receives a prop called **greetHandler** from its parent component.

Child Component

```
import React from 'react'
function ChildComponent(props) {
  return (
    <div>
      <button onClick={props.greetHandler}>Greet Parent</button>
    </div>
  )
}
export default ChildComponent
```

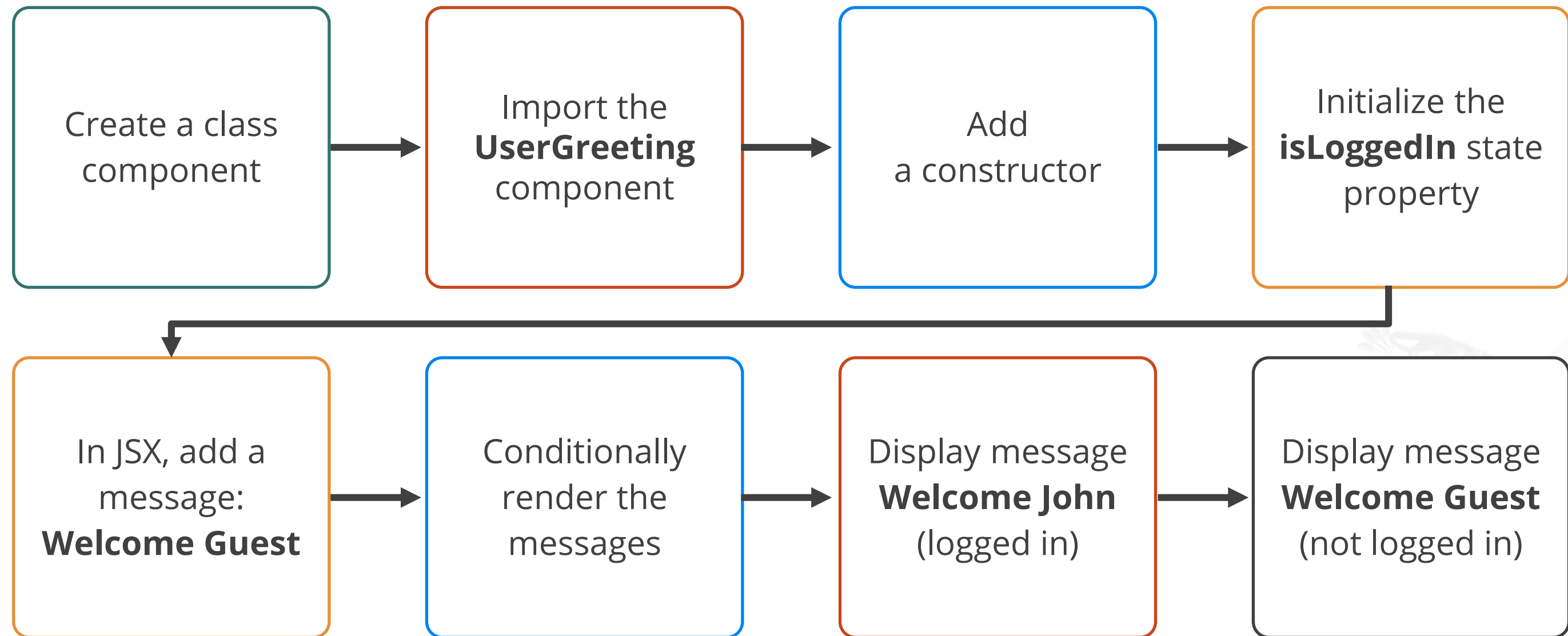
Conditional Rendering in HTML and JavaScript

Conditional rendering is a process where different components are rendered based on certain conditions or values.



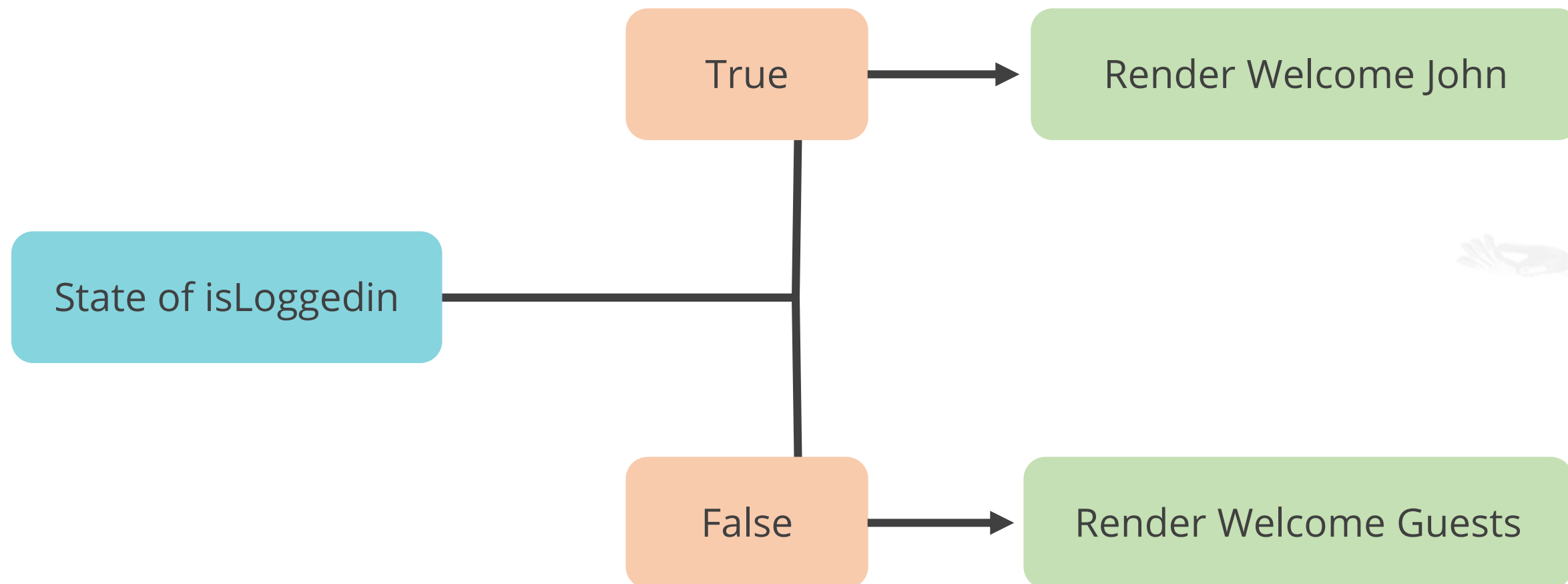
Conditional Rendering: If/Else

To create a new file UserGreeting.js:



Conditional Rendering: If/Else

When the app is run and viewed in the browser, it will render either **Welcome John** or **Welcome Guest** based on the value of **isLoggedIn** in the **UserGreeting** component's state.



Conditional Rendering: If/Else: Code

The conditions are checked in conditional rendering, and based on the results, different components are rendered. Here is a program that highlights conditional rendering by using the **If/Else** code:

```
1  import React, { Component } from 'react';
2
3  class UserGreeting extends Component {
4    constructor(props) {
5      super(props)
6      this.state = {
7        isLoggedIn: false
8      }
9    }
10
11   render() {
12     if(this.state.isLoggedIn) {
13       return (
14         <div>Welcome John</div>
15       )
16     } else {
17       return (
18         <div>Welcome Guest</div>
19       )
20     }
21   }
22 }
```



Conditional Rendering: If/Else: Code

Different components can be rendered dynamically in conditional rendering based on the logic.
Here is the remaining part of the code:

```
1  import './App.css';
2  import UserGreeting from './components/UserGreeting';
3
4  function App() {
5    return(
6      <div className="App">
7        <UserGreeting/>
8      </div>
9    );
10 }
11
12 export default App;
```



Conditional Rendering: Element Variable

Element variables are used for the conditional rendering of components in JavaScript.



The variable containing the element is returned in the JSX.

Example:

if (this.state.isLoggedIn), message = div tag with text Welcome John, else message = div tag with text Welcome Guest

Conditional Rendering: Element Variable Code

Conditional rendering using element variables allows developers to display elements selectively based on conditions. Here is an example that shows conditional rendering:

```
1  import React, { Component } from 'react';
2
3  class UserGreeting extends Component {
4    constructor(props) {
5      super(props)
6      this.state = {
7        isLoggedIn: false
8      }
9    }
10
11    render() {
12      if(this.state.isLoggedIn) {
13        return (
14          <div>Welcome John</div>
15        )
16      } else {
17        return (
18          <div>Welcome Guest</div>
19        )
20      }
21    }
22  }
```



Ternary Conditional Operator

The ternary operator can be used within the JSX by adding a return statement and using a conditional operator with parentheses to display the appropriate element. Here is a code that explains this aspect:

```
1  import React, { Component } from 'react';
2
3  class UserGreeting extends Component {
4    constructor(props) {
5      super(props)
6      this.state = {
7        isLoggedIn: false
8      }
9    }
10
11   render() {
12     if(this.state.isLoggedIn) {
13       return (
14         <div>Welcome John</div>
15       )
16     } else {
17       return (
18         <div>Welcome Guest</div>
19       )
20     }
21   }
22 }
```



Ternary Conditional Operator

The text should appear in the browser once the code is executed.



Short Circuit Operator

This approach is a specific case of the ternary operator when rendering either something or nothing.

```
1 import React, { Component } from 'react'
2
3 class UserGreeting extends Component {
4   constructor(props) {
5     super(props)
6     this.state = {
7       isLoggedIn: true
8     }
9   }
10  render(){
11    return this.state.isLoggedIn && <div>Welcome John</div>
12  }
13 }
14
15 export default UserGreeting
```

To render **Welcome John**:

Demo with Conditional Rendering



Duration: 10 min.

Problem Statement:

You are given a project to create a counter application in React to increase the counter value on each click. However, the value displayed on the web page should always be an even number.

ASSISTED PRACTICE

Assisted Practice: Guidelines

The steps to be followed are:

1. Create a class component
2. Create an **onClick** function to update the counter value
3. Create a render method to return the counter value only if it is even



Key Takeaways

- The component-based architecture of React breaks down an application into small, individual parts. These are made into complex UIs.
- There are two components: a stateless functional component and a stateful class component.
- React supports the features of stateful components in functional components. It allows state features without using classes.
- Prop is the optional input that a component can accept.
- Destructuring pulls data from an array or object and assigns it to its variables.



Demonstrable Project with Event Handling and Components

Duration: 35 Min.

Project Agenda:

Create a Notes application using React so that when the user types in the text and clicks on **Add Notes**, the text gets displayed below.

Description:

Create a component that accepts an array as input, creates an array with text from each note, and builds a new state from the present state.

Perform the following:

1. Create a component that accepts an array as input and returns the notes
2. Create an array that has the text of each note
3. Add the option to add tasks to the list by using the useState hook
4. Add an input field and a button to the app to add more notes



TECHNOLOGY

Thank You