

TECHNOLOGY



ReactJS

Rendering, LifeCycle, and React Hook Concepts



Learning Objectives

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

- 🕒 Use lists for creating interactive user interfaces
- 🕒 Work with `map()` method
- 🕒 Add CSS and styling to React components
- 🕒 Handle forms in React
- 🕒 Explain the different phases of a component's lifecycle
- 🕒 Use `useState` Hook



A Day in the Life of a ReactJS Developer

Sarah is a budding ReactJS Developer who has recently started working on a new project. Her task is to make a web application that shows a list of products for an e-commerce site.

Sarah's web application must:

- Handle the list of products using React
- Iterate an array
- Develop a new array

Along with these tasks, the web application must have a form component that allows users to search for products by name or filter them by price range.

The following slides show how to do these activities.



List Rendering

List Rendering: An Overview

List rendering displays a set of related data as a list on a web page.

It maps an array of data to create a new array of components.

It includes related data, such as names, products, books, and more.

List Rendering

React provides a simple and efficient way to render lists using the **map() method**.

```
const books = [  
  { title: "Time Machine",  
    author: "H.G. Wells" },  
  { title: "Origin of  
Species", author: "Charles  
Darwin" },  
  { title: "The Odyssey",  
    author: "Homer" }  
];
```

Example:

An array of objects representing a list of books.

Create a New List

To iterate through the current array, use the **map()** method as shown below:

```
const bookList =  
books.map( (book) =>  
  {  
    return (  
      <li>  
        <h3>{book.title}</h3>  
        <p>by {book.author}</p>  
      </li>  
    );  
  } );
```

This code will create a new array of React components, each representing an item in the list of books.



Create a New List

The components can then be displayed on the screen as an unordered list by using the following:

```
<ul>
  {bookList}
</ul>
```

This will display the list of books on the web page, with each book's title and author displayed in the web page's list item.

Lists and Keys

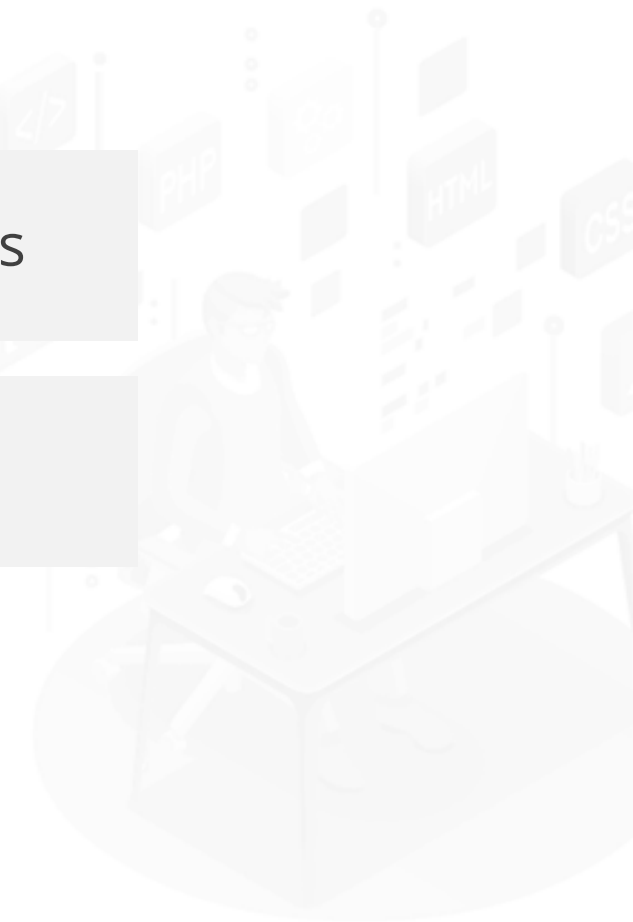
Lists and Keys

Lists

An array of related data rendered as a group of React components

Keys




A special attribute used when rendering lists to help identify each component in the list



Use of Keys

The key prop should be a unique string or number that uniquely identifies a list item among the siblings of the components in the list.

React uses the key prop to perform the following actions:

-  Define an event handler function
-  Speed up rendering
-  Determine which components have been added, removed, or relocated

Use of Keys: An Example

Use the **map() method** to iterate through the item array and generate a fresh array of React components.

```
const items = [  
  { id: 1, name: "Item 1"  
},  
  { id: 2, name: "Item 2"  
},  
  { id: 3, name: "Item 3" }  
];
```



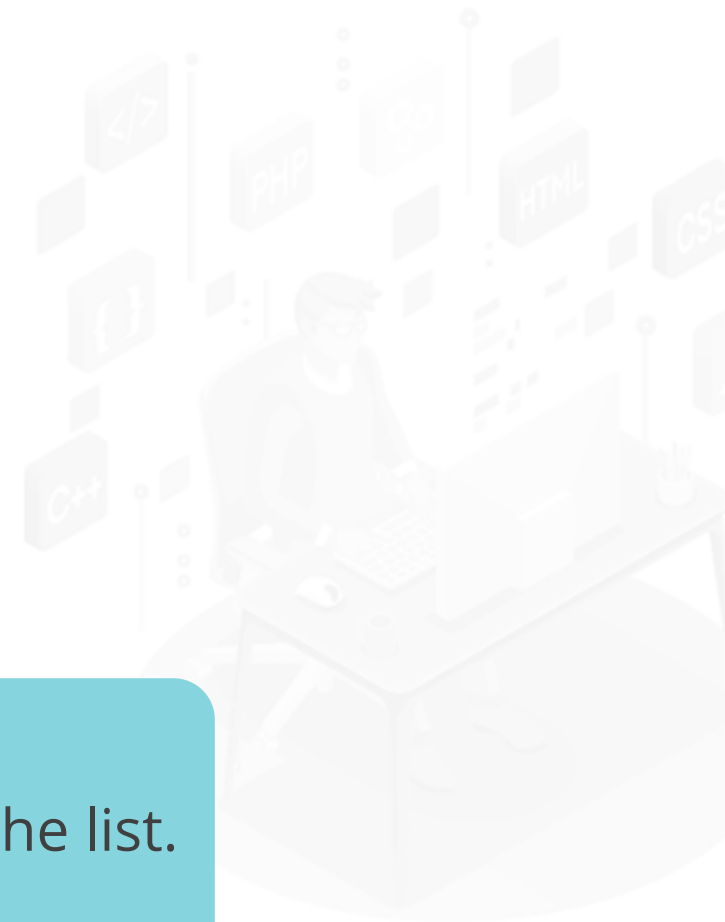
Use of Keys: An Example

In the following code, keys are used to sufficiently update the DOM:

```
const itemComponents = items.map(item => (  
  <li key={item.id}>  
    {item.name}  
  </li>  
));
```

In the code,

- The ID property serves as the key prop for its corresponding component in the list.
- When a list item is added, removed, or reordered, the DOM is updated.
- Without keys, React may need to re-render the entire list on every update.



Demo with List Rendering



Duration: 10 min.

Problem Statement:

You are given a project to implement a component that renders the list of books as an unordered list and uses keys to optimize the rendering performance.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Define a booklist component to accept an array of books as a prop
2. Assign unique key prop to each component of the book
3. Render the array of component as an unordered list



Styling and CSS Basics

Styling and CSS are used to enhance the visual appearance and layout of components.

CSS and styling can be added in the following ways:

Inline Styles

CSS Modules

CSS in JS Library

Styling and CSS Basics

This example defines a style object with CSS properties as keys and values.

```
function MyComponent() {  
  const styles = {  
    backgroundColor: "blue",  
    color: "white",  
    padding: "10px",  
    borderRadius: "5px"  
  };  
  return <div style={styles}>Hello, world!</div>;  
}
```

Inline Styles are defined for a component using JavaScript Object, and are applied directly to the component's HTML elements.



Styling and CSS Basics

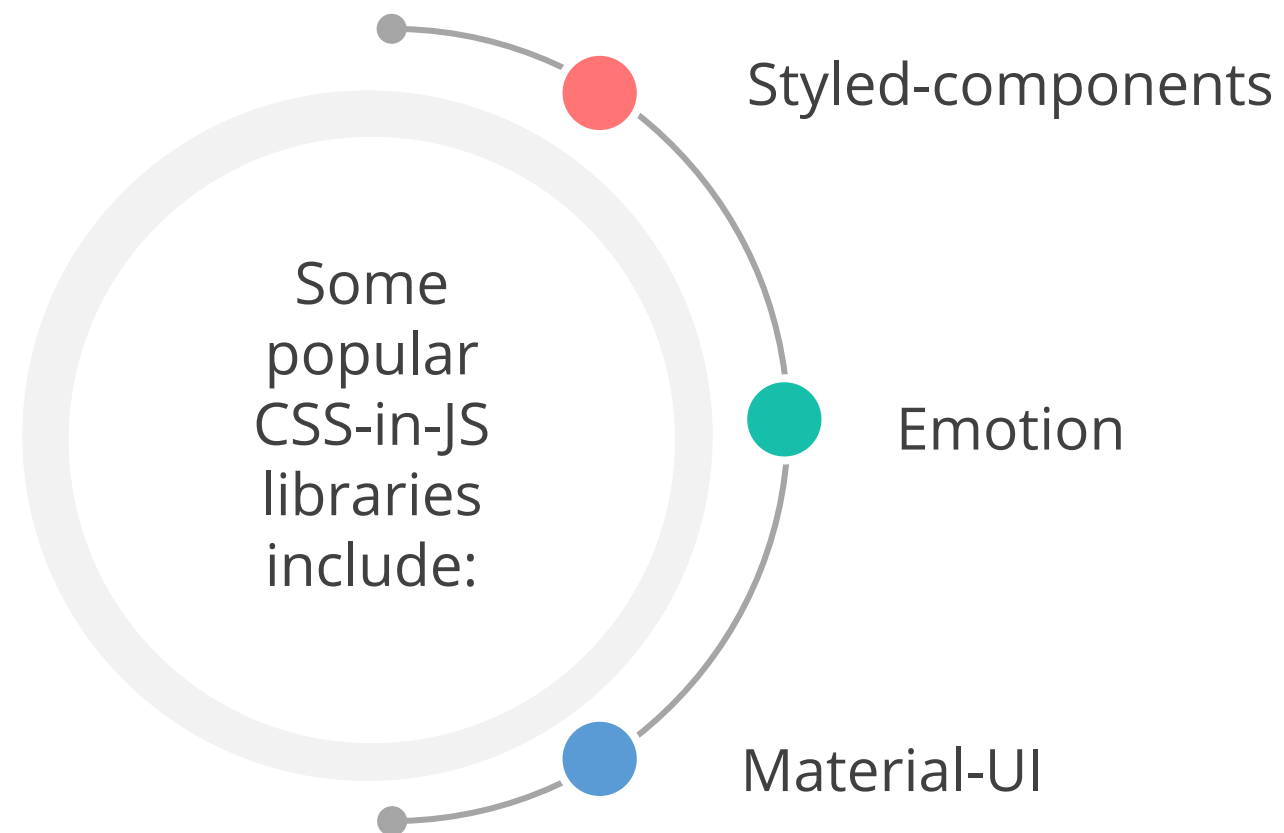
CSS modules define CSS styles in a separate file and import them into components as objects. It also defines class names that are scoped locally to the component level.

In this example, a CSS file called **MyComponent.module.css** is imported that has styles for the container class.

```
import styles from "../MyComponent.module.css";  
function MyComponent() {  
  return <div className={styles.container}>Hello, world!</div>;  
}
```

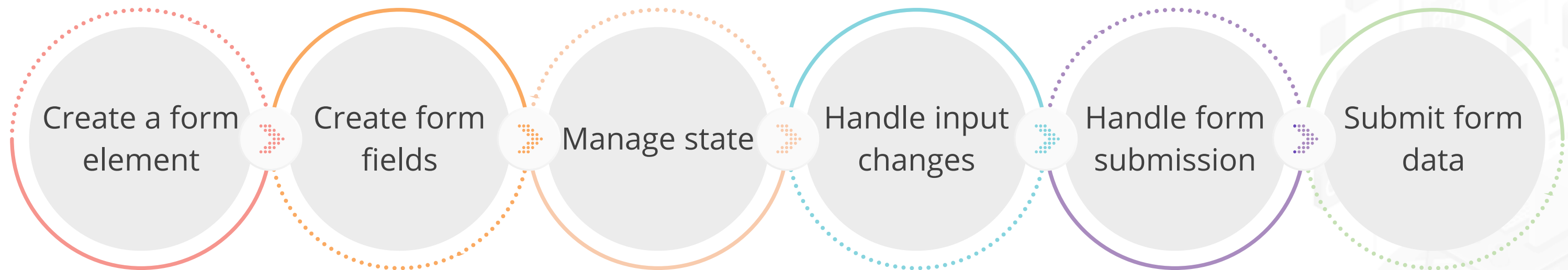
Styling and CSS Basics

The **CSS-in-JS Library** provides dynamic styling, server-side rendering, and other functionality by simply defining CSS styles in JavaScript code.



Basics of Form Handling

The basic steps involved in handling a form in React are as follows:



Form Handling: An Example

Here's an example of basic form handling in React:

```
import { useState } from 'react';
function MyForm() {
  const [fullName, setFullName] = useState('');
  const [userEmail, setUserEmail] = useState('');
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(`Full Name: ${fullName}, Email: ${userEmail}`);
  };
  // Rest of the component code...
  return (
    // JSX code for your form...
  );
}
```

A React developer must use the useState Hook to manage the state of two form fields: name and email.

Form Handling: An Example

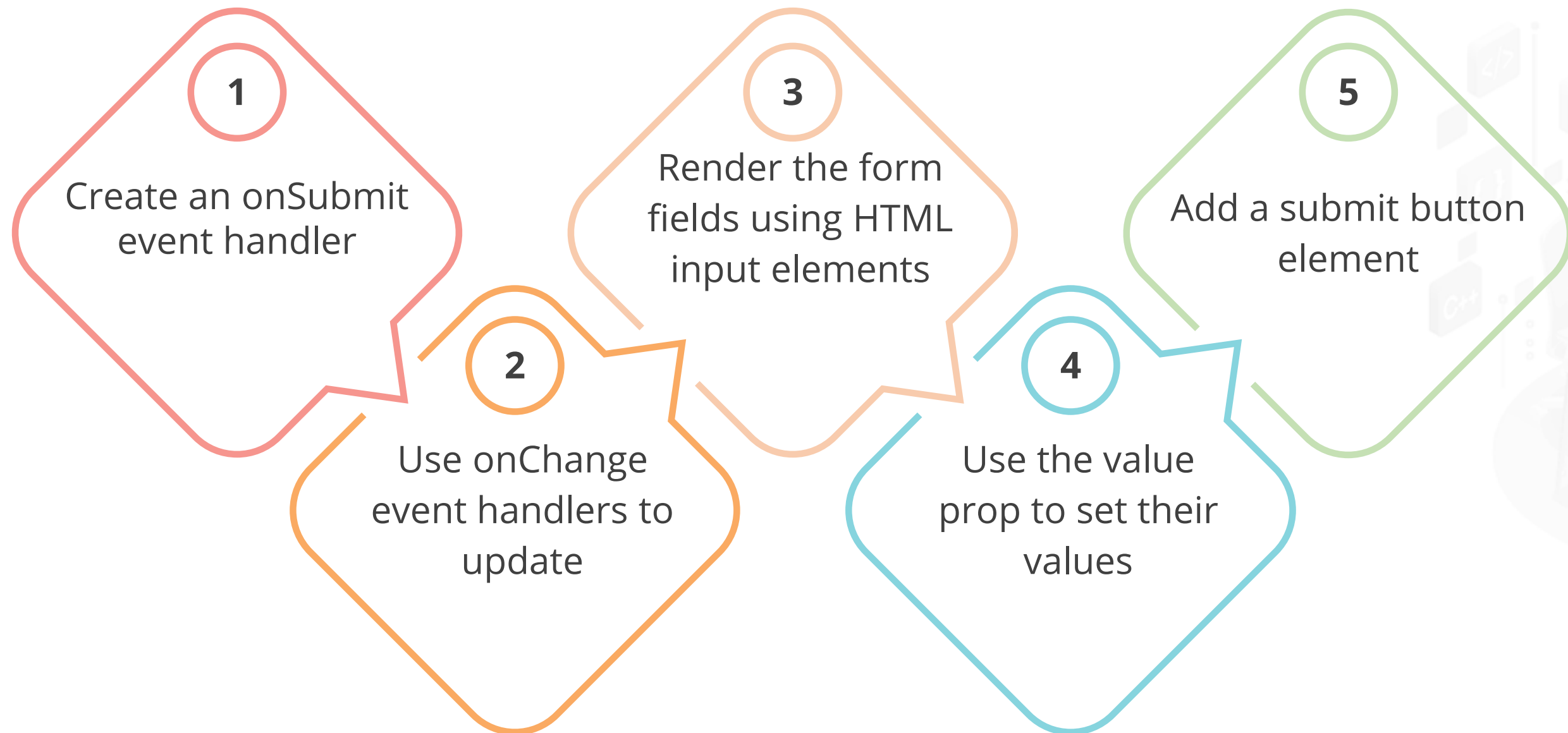
This code represents a form with input fields for name and email, allowing users to submit their information.

```
return (  
  <form onSubmit={handleSubmit}>  
    <label>  
      Name:  
      <input type="text"  
name="name" value={name}  
onChange={ (e) =>  
setName(e.target.value)} />  
    </label>  
    <label>
```

```
Email:  
      <input type="email"  
name="email" value={email}  
onChange={ (e) =>  
setEmail(e.target.value)} />  
    </label>  
    <button  
type="submit">Submit</button>  
  </form>  
); }
```

Form Handling: An Example

This set of steps involves creating an event handler and updating and rendering form fields for submitting a form.



Demo with Form Handling



Duration: 20 min.

Problem Statement:

You are given a project to create a simple form with three input fields: name, email, and message.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a **BasicFormHandling** function
2. Create state handlers to set the **formValues** and errors
3. Create a function to handle changes
4. Create a function handle for submission
5. Render the components on the page



Lifecycle of React Component

Component Lifecycle Methods

Component Lifecycle Methods in React are predefined methods that are invoked at different stages of a component's existence.

These methods have a range of advantages, such as:

Control and Flexibility

Initialization and Cleanup



Component Lifecycle Methods

Here are some more advantages of Component Lifecycle Methods:

Dynamic Updates

Error Handling

Integration with Third-Party Libraries

Legacy Support

Component Mounting Lifecycle Methods

Component Mounting refers to the process of creating an instance of a component and inserting it into the DOM.

1

constructor():

This method is called when the component is first initialized.

2

render():

This method is responsible for rendering the component's JSX and is required for every component.

3

componentDidMount():

This method is called after the component is inserted into the DOM.

Component Mounting Lifecycle Methods

There are many component-updating lifecycle methods in React, such as:

1

shouldComponentUpdate(nextProps, nextState):

This method determines component updates based on changes in props or state.

2

componentWillUpdate(nextProps, nextState):

This method prepares for the update.

3

render():

This method re-renders the component with updated data.

4

componentDidUpdate(prevProps, prevState):

This method is used to update the component's state.

Demo with All the Lifecycle Methods



Duration: 20 min.

Problem Statement:

You are given a project to create a React program to implement a timer.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Define a timer component with a state property of seconds
2. Set the initial value of the second state
3. Use the **componentDidMount** method to setup the interval in seconds
4. Avoid memory leaks using **componentWillUnmount**
5. Display seconds value in **render()** methods



Fragments

A Fragment is a feature that allows grouping a list of child elements without creating an additional DOM node.

Consider the given code:

```
render() {  
  return (  
    <div>  
      <h1>Heading</h1>  
      <p>Paragraph</p>  
    </div>  
  );  
}
```

In this code example, the **render() method** generates a JSX expression as its output.



Fragments: Uses

The code given below shows how a Fragment can be used to achieve the same result:

```
render() {  
  return (  
    <>  
      <h1>Heading</h1>  
      <p>Paragraph</p>  
    </>  
  );  
}
```

In the code, the `<>` and `</>` tags represent an empty Fragment that wraps the `h1` and `p` elements.

Fragments: Uses

Fragments are also used with keys to provide a stable identity for elements across updates. For example:

```
render() {  
  return (  
    <>  
      {this.props.items.map(item => (  
        <React.Fragment key={item.id}>  
          <h1>{item.title}</h1>  
          <p>{item.content}</p>  
        </React.Fragment>  
      ))}  
    </>  
  );  
}
```

In this code, we use a fragment with a key to render a list of items.



Pure Components

A Pure Component only re-renders when its props or state change.

In this example, MyComponent extends PureComponent, which automatically implements the **shouldComponentUpdate()** method.

Example of a pure component:

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

class MyComponent extends PureComponent {
  render() {
    return <div>{this.props.title}</div>;
  }
}
```



Pure Components

Some important points:

The **shouldComponentUpdate()** method is called whenever props or state change.

The method compares the previous and current props and state.

The pure components only work if the props and state are immutable.

Pure components can be a useful optimization technique to improve performance.

Demo with Pure Components



Duration: 10 min.

Problem Statement:

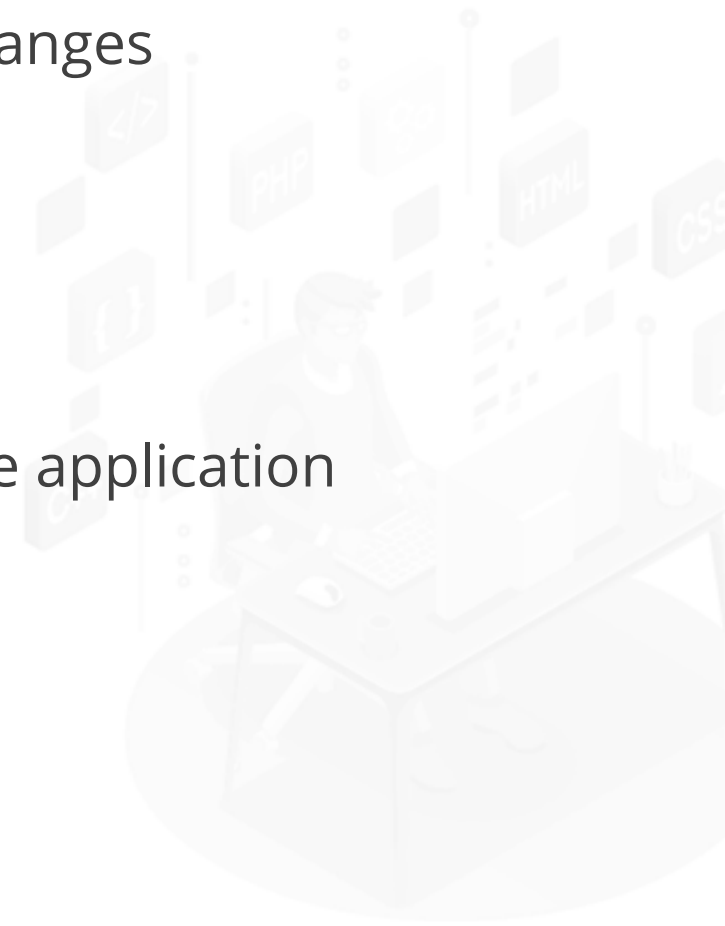
You are given a project to create a component called Greetings that takes in a name prop and returns a greeting message that says Hello, {name}! The component should only re-render when the name of the prop changes.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a pure function component that only re-renders when the **name prop** changes
2. Define an App component that uses the Greetings component
3. Define button to change the name of the prop
4. Import it and pass in the name prop to use this component in another part of the application



useState Hook

React Hooks: Introduction

React Hooks enable the use of state and other features in functional components, eliminating the need for class components.

Here are some rules to follow when implementing React Hooks:

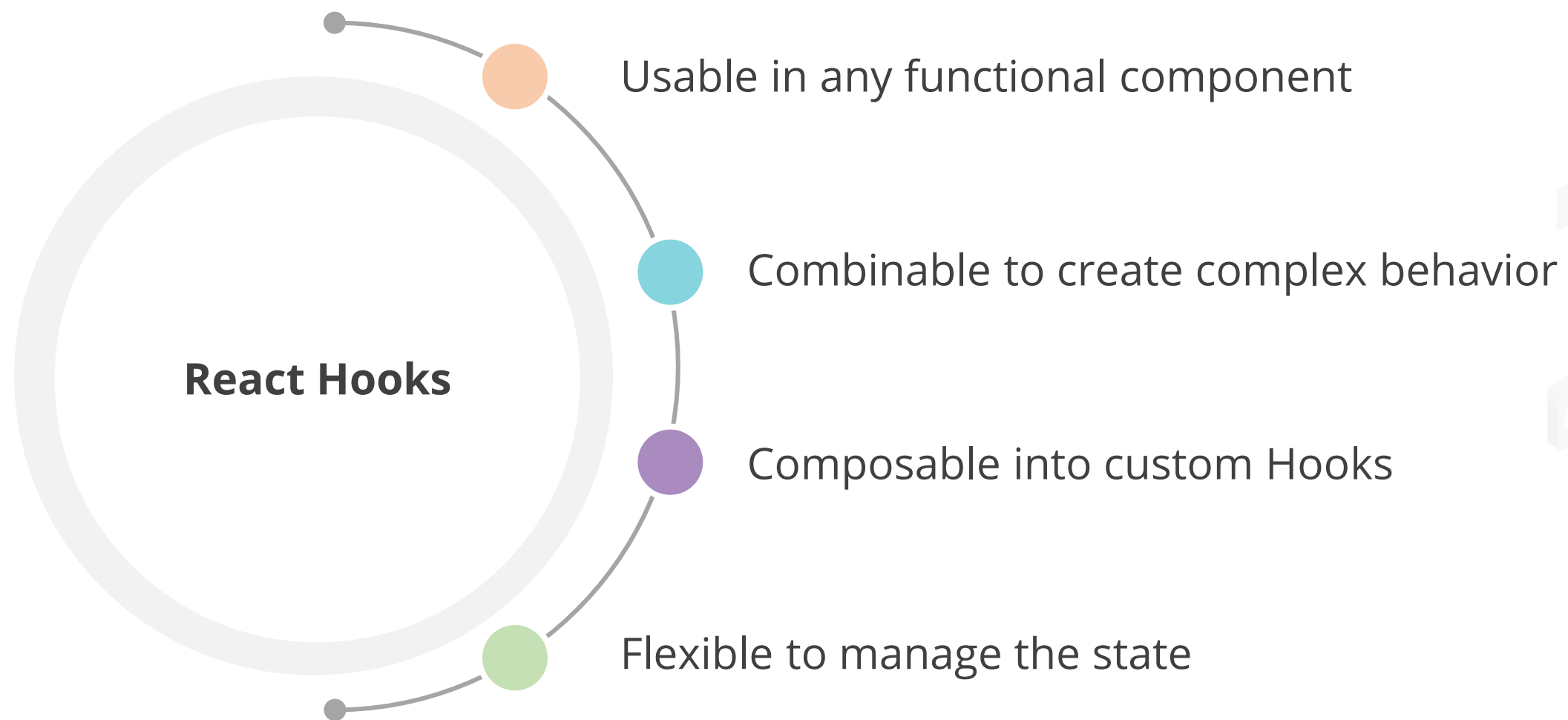
Hooks can be used only at the top level.

It can be called only inside the functional components.

It must always be unconditionally called.

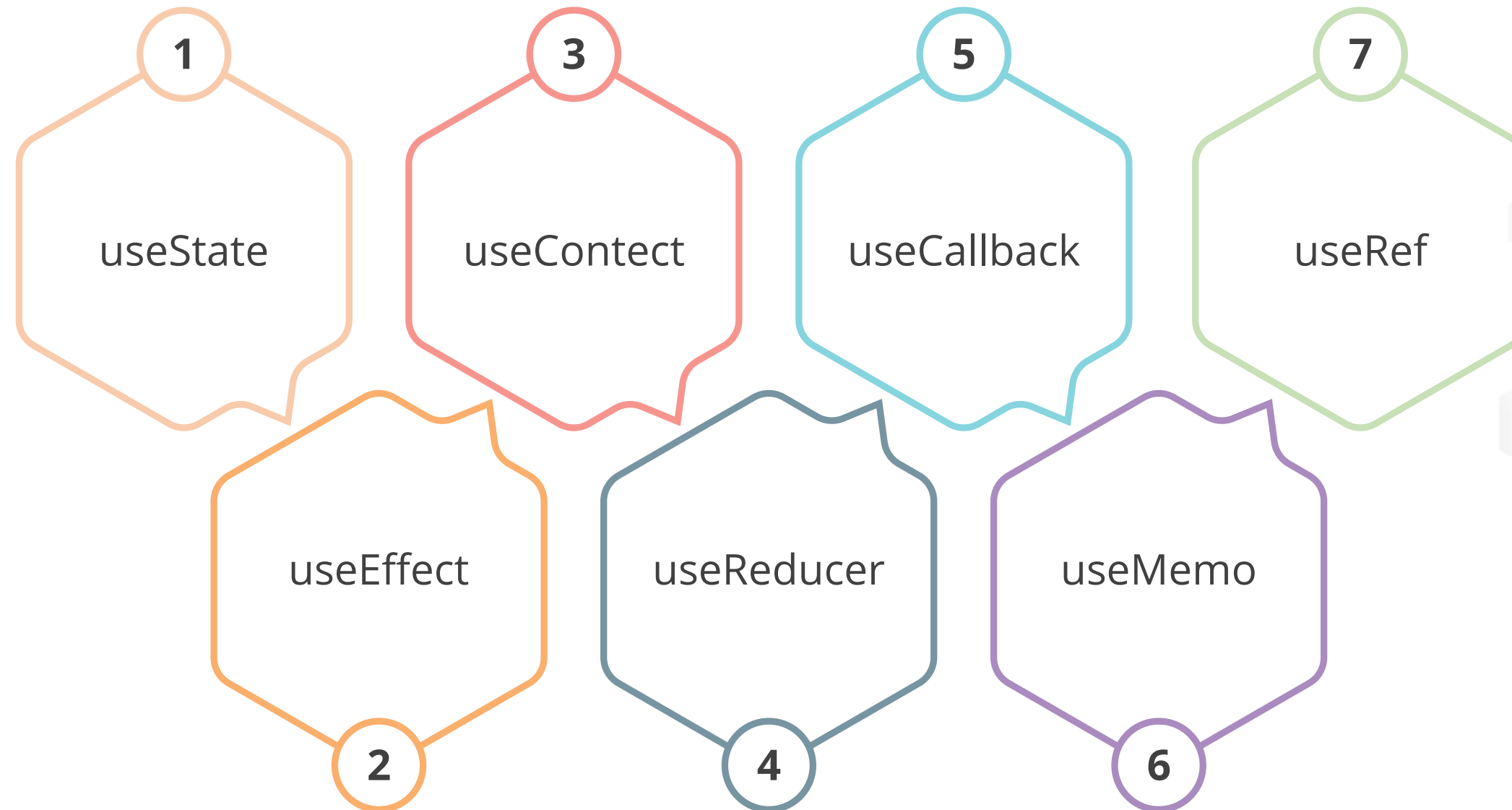
React Hooks: Features

The salient features of React Hooks are as follows:



React Hooks: Examples

Here are the following built-in Hooks that allow developers to manage the state:



useState Hook: Example

This example uses **useState** to define a state variable called count and two event handlers to update it, with **setCount** replacing the previous value of count.

```
import React, { useState } from 'react';
const Counter = () => {
  const [counterValue, setCounterValue] =
    useState(0);
  const handleIncrement = () => {
    setCounterValue(counterValue + 1);
  };
  const handleDecrement = () => {
    setCounterValue(counterValue - 1);
  };
}
```

```
return (
  <div>
    <p>Count: {counterValue}</p>
    <button
      onClick={handleIncrement}>+</button>
    <button onClick={handleDecrement}>-
    </button>
  </div>
);
export default Counter;
```

useState Hook: Example

This example consists of an App component that renders the **counter** component, which updates the count state variable when the + or - buttons are clicked.

```
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <Counter />
    </div>
  );
};

export default App;
```



useState: Updating State

To update state based on its previous value in React using **useState**, use the functional form of **setState**, which takes a function as its argument. Here is an example:

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

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

```
    const handleIncrement = () => {
      setCount(prevCount => prevCount + 1);
    };

    const handleDecrement = () => {
      setCount(prevCount => prevCount - 1);
    };
  };
}
```

In the example, the **counter** component uses **useState** to manage the count state and event handlers to update it with the functional form of **setCount**.

useState: Updating State

This code renders a counter component in React, displaying the current count value and providing buttons to increment and decrement the count.

```
return (  
  <div>  
    <p>Count: {count}</p>  
    <button onClick={handleIncrement}>+</button>  
    <button onClick={handleDecrement}>-</button>  
  </div>  
);  
};  
  
export default Counter;
```



useState: Defining Object State Variables

Object State Variables can be defined in React using the **useState** Hook by passing an object as the initial state value. Here is an example:

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

const Form = () => {
  const [user, setUser] = useState({
    name: '',
    email: '',
    password: '',
  });

  const handleChange = event => {
    setUser({
      ...user,
      [event.target.name]: event.target.value,
    });
  };
};
```



useState: Defining Object State Variables

The code defines a **handleSubmit** function that prevents default form submission. It also logs the user object and renders a form with an input field.

```
const handleSubmit = event => {  
  event.preventDefault();  
  console.log(user);  
};  
  
return (  
  <form onSubmit={handleSubmit}>  
    <label htmlFor="name">Name</label>  
    <input type="text" id="name" name="name" value={user.name} onChange={handleChange} />  
  );  
};
```


useState: Defining Object State Variables

Here, the code includes input fields for email and password, with corresponding labels, and a submit button, all contained within a form component.

```
<label htmlFor="email">Email</label>
  <input type="email" id="email"
name="email" value={user.email}
onChange={handleChange} />

  <label
htmlFor="password">Password</label>
  <input type="password"
id="password" name="password"
```

```
value={user.password}
onChange={handleChange} />

    <button
type="submit">Submit</button>
  </form>
  );
};

export default Form
```

useState: Defining Array State Variables

This code uses the **useState** Hook to define Array State Variables by passing an array as the initial state.

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

const List = () => {
  const [items, setItems] =
    useState(['Item 1', 'Item 2', 'Item 3']);

  const handleClick = () => {
    setItems([...items, `Item
    ${items.length + 1}`]);
  };

  return (
    <div>
```

```
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <button onClick={handleClick}>Add
      Item</button>
    </div>
  );
};

export default List;
```

The code defines a List component that maintains a state variable called **items** as an array of strings.

Demo with useState Hook



Duration: 10 min.

Problem Statement:

You are given a project to build a new React component called **Counter** using the useState Hook.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new React component called **Counter**
2. Define a state variable called count using the **useState Hook**, with an initial value of 0
3. Render a heading that displays the current value of the count variable
4. Render two buttons: one that increments the count by 1, and another that decrements the count by 1
5. Add functionality to the buttons so that when they are clicked, the count variable is updated
6. Make sure the count variable cannot go below 0

Key Takeaways

- The `map()` method allows iteration of an array of data to create a new array of React components based on that data.
- There are several ways to add styling and CSS to the React components, such as Inline styles, CSS modules, and CSS-in-JS Libraries.
- A Fragment is a feature in React that allows users to group a list of child elements without creating an additional DOM node.
- In React, a Pure Component is a component that only re-renders when its props or state change.
- The `useState` Hook is a built-in React Hook that allows users to add a state to the functional components.



Lesson-End Project: React Hooks

Duration: 30 Min.

Problem Agenda:

Create a project on the **To-Do List** application using the concepts of Hooks and styling that are explained in this session

Description:

This application ensures that users can add tasks to a ToDo List. They can also remove tasks or mark tasks as completed. The application uses the useState Hook to manage the state of the ToDo List, and the input field for adding new tasks.

Perform the following:

1. Create a functional component for the To-Do List
2. Use Hooks like useState and useEffect
3. Implement the logic for adding, deleting, and updating tasks
4. Apply styling to the To-Do List component using CSS or a styling library
5. Test the To-Do List application by adding, deleting, and updating tasks



TECHNOLOGY

Thank You