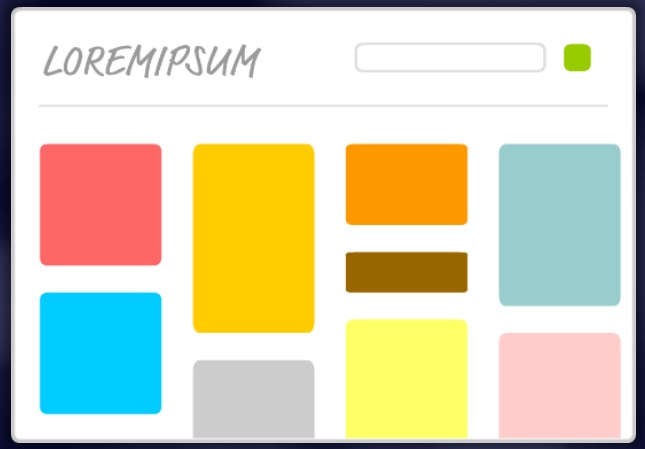
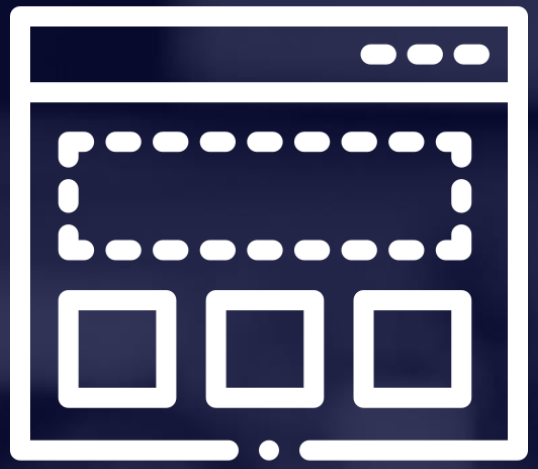
React Components – Basic Idea



How to Compose in React ?

Table of Contents



Components Overview

Syntax, Functional and Class Components

Components Overview

- Components let you
  - Split the UI into **independent** and **reusable pieces**
  - Think about **isolation**
- React let you define components as
  - **Functions**
  - **Classes**

Functional Component

- **Functional component** is a JS function which
  - Accepts single argument called **props** (object with data)
  - Returns a **React Element**

```
function Person(props){
  return <div>My name is {props.name}</div>
}
```

Class Component

- To define a **React component class**, you need to extend **React.Component**

```
class Person extends React.Component {
  render() {
    return <h1>My name is {this.props.name}</h1>
  }
}
```

- The only method you must define is called **`render()`**

Component Syntax

- Names always start with **UpperCase**
- Tags always must be **closed**
- **Information** is passed via **props**

```
<Dropdown> A dropdown list
  <UserHead name="homeHeader" />
  <Menu>
    <MenuItem>Do Something</MenuItem>
    <MenuItem>Do Something Fun!</MenuItem>
  </Menu>
</Dropdown>
```

Component Props and State

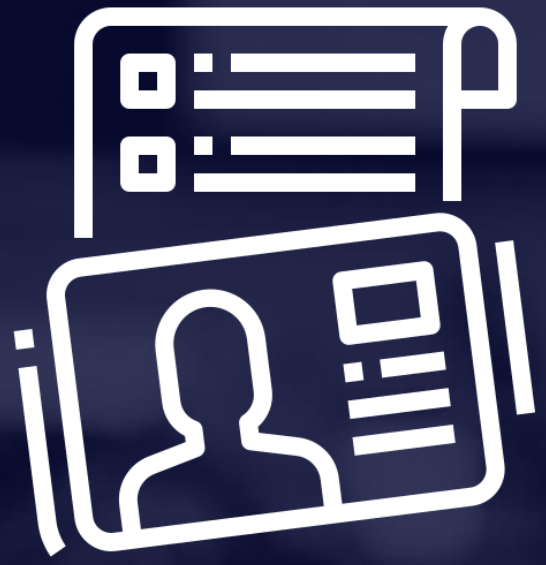
Overview


Props and State Overview

- In React **this.props** and **this.state** represent the rendered values
- Both are plain JavaScript **objects**
- Both hold information that influences the output of render

Props and State Overview

- They are different in one important way
  - **props** get passed to the component (like function params)
  - **state** is managed within the component (like local variables)



Component Props

Passing Data, Access and Usage

Component Props

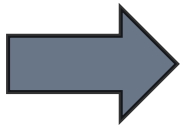
- **props** is short for **properties**
  - are received from above (parent)
  - **immutable** as far as the component receiving them is concerned
- A component **cannot change** its own props, but it is responsible for putting together the props of its child components

Passing Props to Nested Components

- We use props to **pass data** from parent to child

```
const BookList = () => {
  return (
    <ul>
      <Book
       title="IT"
       author="Stephen King"
       price="20"
      />
      <Book
       title="The Hunger Games"
       author="Suzanne Collins"
       price="10"
      />
    </ul>
  );};
```

Prop name should start with lowercase letter



Use className to set css classes

```
const Book = (props) => {
  return (
    <li className="book">
      <div>{props.title}</div>
      <div>{props.author}</div>
      <div>{props.price}</div>
    </li>
  );
};
```

Passing Props in Class Components

- Pass props inside **constructor** and use **this** to access them

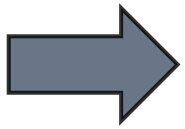
```
class Book extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <li className="book">
        <div>{this.props.title}</div>
        <div>{this.props.author}</div>
        <div>{this.props.price}</div>
      </li>
    );
}}
```

Pass props to base component constructor

Children Property

- Use **children** property to access information between **opening** and **closing** tags

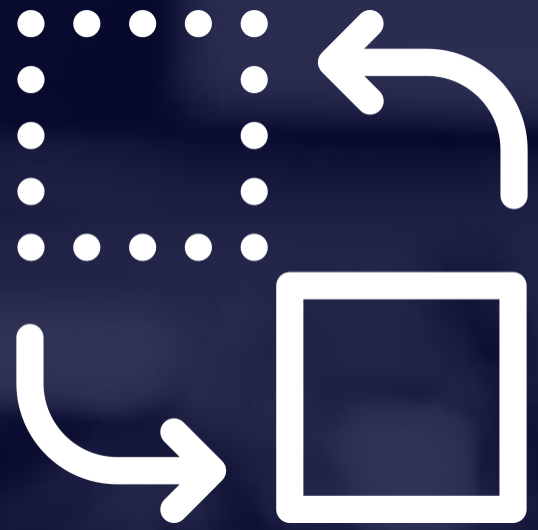
```
const BookList = () => {
  return (
    <ul>
      <Book
        title="IT"
        author="Stephen King"
        price="20">
        <span>
          Some value here
        </span>
      </Book>
    </ul>
  );};
```



```
const Book = (props) => {
  return (
    <li className="book">
      …
       <div>{props.children}</div>
    </li>
  );
};
```

Could be plain text or nested HTML

Props Demo



Storing and Modifying Data

Component State

Component State Overview

- The heart of every React component is its "**state**"
  - It determines how the component **renders** and **behaves**
  - State allows you to create components that are **dynamic** and **interactive**

State

- **State** starts with default value when a component mounts
  - after mounts, suffers from **mutations** in time
  - its **serializable**
- Component manages its own state internally

Component State Example

- State holds information that **can change** over time
  - Usually as a result of **user input** or **system events**

```
class Button extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    this.updateCount = this.updateCount.bind(this);
  }
  updateCount() {
    this.setState((prevState) => { count: prevState.count + 1 });
  }
  render() {
    return (<button onClick={this.updateCount}>
      Clicked {this.state.count} times</button>);
  }}
```

Working with States

- State is used only with **class-based components**
  - Default state can be set in the constructor
- To access state

```
console.log(this.state);
```

- State must never be directly modified
  - Use **`this.setState`** instead
  - New state will be **merged** with old state

Working with States

- **`setState()`**
  - schedules an update to a component's state object
  - when state changes, component response by **re-rendering**
- Calls to **setState** are **asynchronous**
  - inside event handlers
  - don't rely on **`this.state`** to reflect the new value **immediately**

Working with States

- Passing **object** or **callback function**

```
class Button extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    this.updateCount = this.updateCount.bind(this);
  }
  updateCount() {
    this.setState((prevState) => { count: prevState.count + 1 });
  }
  render() {
    return (<button onClick={this.updateCount}>
      Clicked {this.state.count} times</button>);
  }}
```

Stateless Component

- **Stateless** Components
  - only props, no state
  - there's no much going on besides the **`render()`**
  - easy to follow and test

```
function Show(props) {
  return (
    <p>{props.value}</p>
  )
}
```

Stateful Component

- **Stateful** Components (State Managers)
  - Both - **props** and **state**
  - They are in charge or client-server communications, processing data and responding to user events
  - Has methods

Stateful Component

```
class Input extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: ""
    };
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }

  render() {...
}
```

Stateful Component (2)

```
constructor(props) { ...
handleChange(event) { ...
render() {
  return (
    <div>
      <input
        name="firstName"
        type="text"
        value={this.state.value}
        onChange={this.handleChange}
      />
      <Show value={this.state.value} />
    </div>
  );
}
```

State Demo

Handling Events

Handling Events

- Handling events with React elements is very similar to handling event on **DOM elements**
- The syntactic differences are:
  - React events are named using **camelCase**
  - With JSX you pass a function as the **event handler**

Handling Events

- When using React you should generally
  - **Not** need to call **`addEventListener`** to add listeners to a DOM element after it is created
  - Just provide a listener when the element is initially rendered

```
<button onClick={this.clickHandler}
  Click me! I'm a counter
</button>
```

Handling Events

- There are two ways to passing arguments to event handlers
  - using **arrow functions**

```
<button onClick={(e) => this.deleteRow(id, e)}>
  Delete Row
</button>
```

  - using **bind**

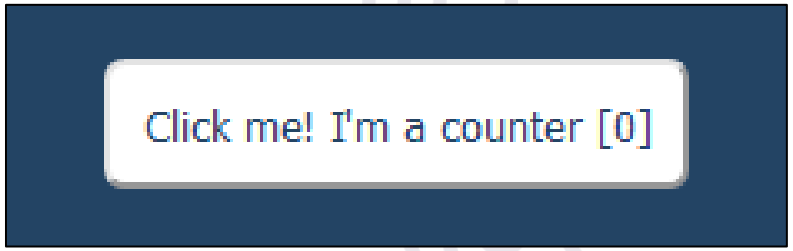
```
<button onClick={this.deleteRow.bind(this, id)}>
  Delete Row
</button>
```

Handling Events

```
clickHandler = () => {
  const currentClicks = this.state.clicks;
  this.setState({ clicks: currentClicks + 1 })
}
```

```
<Button
  clickHandler={() => this.clickHandler()}
  clicks={this.state.clicks}
/>
```

```
<button className="counter"
  onClick={props.clickHandler}>
    Click me! I'm a counter [{props.clicks}]
</button>
```



SyntheticEvent

- Event handlers will be passed instances of **`SyntheticEvent`**
  - It has the same interface as the browser's native event
    - Including **`stopPropagation()`** and **`preventeDefault()`**
    - Except the events work identically across all browsers

```
function onClick(event) {
  console.log(event);
  console.log(event.type);
  const eventType = event.type;
}
```

Event Pooling

- Event **pooling**
  - **SyntheticEvent** object will be reused and all properties will be **nullified** after the event callback has been invoked
  - **cannot access** the event in **async** way
    - It's possible by using **event.persist()**
- **React Supported Events**

Handling Events Demo

Conditional Rendering

Conditional Rendering

- Conditional rendering in React works the same way conditions work in JavaScript using:
  - operators like **if**
  - conditional (**ternary**) operators

Conditional Rendering

- Using **if** operator

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
    if (isLoggedIn) {
      return <UserGreeting />;
    }
  return <GuestGreeting />
}
```

Conditional Rendering

- Using **ternary** operator

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

function Greeting(props) {
  return (
    <div>
      { props.isLoggedIn ? < UserGreeting /> : <GuestGreeting /> }
    </div>
  )}
```

Conditional Rendering Demo

Summary

- **Components** reusable elements
  - **Functional and Class**
- **Props** are used to pass down data
- **State** is used to hold component data
- Handling Events in React
- Conditional Rendering
  - **If** and **ternary** operators

Questions?

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © Kingsland University – https://kingslanduniversity.com


KINGSLAND
UNIVERSITY
THANK YOU