SNS COLLEGE OF ENGINEERING

An Autonomous Institution

Coimbatore-107

19TS601-FULL STACK DEVELOPMENT

UNIT-2

REACT

5. React state - setting State - Async State Initialization

React Class Component State

- React Class components have a built-in state object.
- Refer component constructor section.
- The state object is where you store property values that belongs to the component.
- When the state object changes, the component re-renders.

Creating the state Object

- The state object is initialized in the constructor:

Example

Specify the `state` object in the constructor method:

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

- The state object can contain as many properties as you like:

Example

Specify all the properties your component need:

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
}
```

Using the state Object

- Refer to the state object anywhere in the component by using the this.state.propertyname syntax:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My {this.state.brand}</h1>
        <p>
          It is a {this.state.color}
          {this.state.model}
          from {this.state.year}.
        </p>
      </div>
    );
  }
}

ReactDOM.render(<Car />, document.getElementById('root'));
```

My Ford

It is a red Mustang from 1964.

Changing the state Object

- To change a value in the state object, use the this.setState() method.
- When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

```
import React from 'react';
import ReactDOM from 'react-dom/client';

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

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```


My Ford
It is a red Mustang from 1964.
Change color


My Ford
It is a blue Mustang from 1964.
Change color

- Always use the setState() method to change the state object, it will ensure that the component knows its been updated and calls the render() method (and all the other lifecycle methods).

Lifecycle of Components

- Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.
- The three phases are:
  - Mounting,
  - Updating, and
  - Unmounting.

Mounting

- Mounting means putting elements into the DOM.
- React has four built-in methods that gets called, in this order, when mounting a component:
  - constructor()
  - getDerivedStateFromProps()
  - render()
  - componentDidMount()
- The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor

- The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.
- The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

- The constructor method is called, by React, every time you make a component

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

My Favorite Color is red

getDerivedStateFromProps

- The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.
- This is the natural place to set the state object based on the initial props.
- It takes state as an argument, and returns an object with changes to the state.
- The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

- The getDerivedStateFromProps method is called right before the render method:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow"/>);
```

My Favorite Color is yellow

render

- The render() method is required, and is the method that actually outputs the HTML to the DOM.
- The following example shows a simple component with a simple render() method:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

This is the content of the Header component

componentDidMount

- The componentDidMount() method is called after the component is rendered.
- This is where you run statements that requires that the component is already placed in the DOM.

componentDidMount

- The componentDidMount() method is called after the component is rendered.
- This is where you run statements that requires that the component is already placed in the DOM.

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

My Favorite Color is yellow

At first my favorite color is red, but give me a second, and it is yellow instead

Updating

- The next phase in the lifecycle is when a component is updated.
- A component is updated whenever there is a change in the component's state or props.
- React has five built-in methods that gets called, in this order, when a component is updated:
  - getDerivedStateFromProps()
  - shouldComponentUpdate()
  - render()
  - getSnapshotBeforeUpdate()
  - componentDidUpdate()
- The render() method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

- Also at updates the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.
- This is still the natural place to set the state object based on the initial props.
- The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

- If the component gets updated, the getDerivedStateFromProps() method is called:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow" />);
```


```
/*
This example has a button that changes the favorite color to blue,
but since the getDerivedStateFromProps() method is called,
the favorite color is still rendered as yellow
(because the method updates the state
with the color from the favcol attribute).
*/
```

shouldComponentUpdate

- In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.
- The default value is true.
- The example below shows what happens when the shouldComponentUpdate() method returns false

- Stop the component from rendering at any update:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return false;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

My Favorite Color is red

Change color

- Example:Same example as above, but this time the shouldComponentUpdate() method returns true instead:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  shouldComponentUpdate() {
    return true;
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```


My Favorite Color is red
Change color
My Favorite Color is blue
Change color

render

- The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.
- The example below has a button that changes the favorite color to blue:
- Click the button to make a change in the component's state:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```


My Favorite Color is red
Change color


My Favorite Color is blue
Change color

getSnapshotBeforeUpdate

- In the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.
- If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

- The example below might seem complicated, but all it does is this:
- When the component is mounting it is rendered with the favorite color "red".
- When the component has been mounted, a timer changes the state, and after one second, the favorite color becomes "yellow".
- This action triggers the update phase, and since this component has a getSnapshotBeforeUpdate() method, this method is executed, and writes a message to the empty DIV1 element.
- Then the componentDidUpdate() method is executed and writes a message in the empty DIV2 element:

Example

- Use the getSnapshotBeforeUpdate() method to find out what the state object looked like before the update:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
    "Before the update, the favorite was " + prevState.favoritecolor;
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
```

```
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="div1"></div>
      <div id="div2"></div>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

componentDidUpdate

- The componentDidUpdate method is called after the component is updated in the DOM.
- The example below might seem complicated, but all it does is this:
- When the component is mounting it is rendered with the favorite color "red".
- When the component has been mounted, a timer changes the state, and the color becomes "yellow".
- This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

- The componentDidUpdate method is called after the update has been rendered in the DOM

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
```

```
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="mydiv"></div>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header />);
```

Unmounting

- The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.
- React has only one built-in method that gets called when a component is unmounted:
  - componentWillUnmount()

componentWillUnmount

- The componentWillUnmount method is called when the component is about to be removed from the DOM.

- Click the button to delete the header:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
```

My Favorite Color is yellow

The updated favorite is yellow

React Props

- Props are arguments passed into React components.
- Props are passed to components via HTML attributes.
- props stands for properties.

React Props

- React Props are like function arguments in JavaScript and attributes in HTML.
- To send props into a component, use the same syntax as HTML attributes:

Example

Add a "brand" attribute to the Car element:

```
const myElement = <Car brand="Ford" />;
```

The component receives the argument as a `props` object:

Example

Use the brand attribute in the component:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}
```

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

const myElement = <Car brand="Ford" />;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

I am a Ford!

Pass Data

- Props are also how you pass data from one component to another, as parameters.
- React Props are read-only! You will get an error if you try to change their value.
- Send the "brand" property from the Garage component to the Car component:

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Who lives in my Garage?

I am a Ford!

- If you have a variable to send, and not a string as in the example above, you just put the variable name inside curly brackets:

Example

- Create a variable named carName and send it to the Car component:

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  const carName = "Ford";
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carName } />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

- Or if it was an object:

Example

- Create an object named carInfo and send it to the Car component:

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand.model }!</h2>;
}

function Garage() {
  const carInfo = { name: "Ford", model: "Mustang" };
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carInfo } />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Who lives in my Garage?

I am a Mustang!

Thank You