Subject-Web Application Development

Class-10

[Events in React]

Content

Listening and reacting to events

Event properties

Meet synthetic events

Doing stuff with event properties

More eventing shenanigans

You can’t directly listen to event on components

Listen to regular DOM elements

The meaning of ‘this’ inside the event handles

React…Why? Why ?!

Browser compatibility

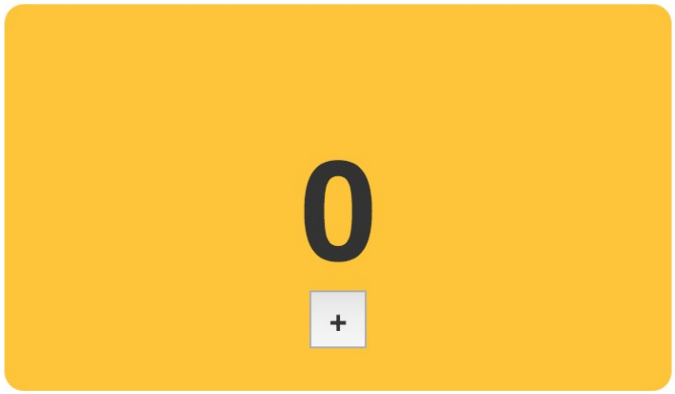
Improved performance

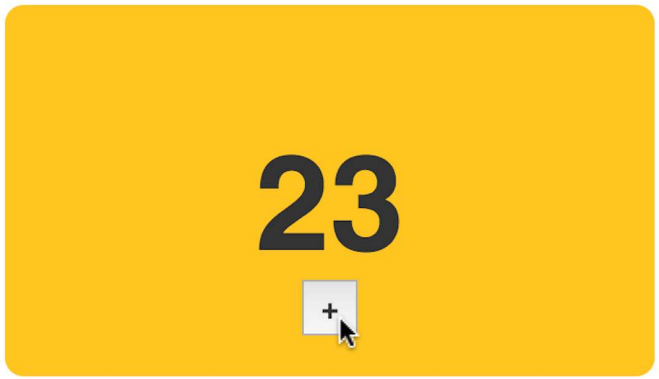
Conclusion

Listening and Reacting to Events

The easiest way to learn about events in React is to actually use them, and that's exactly what we are going to! To help with this, we have a simple example made up of a counter that increments each time you click on a button. Initially, our example will look like this as shown in first figure:

Each time you click on the plus button, the counter value will increase by 1. After clicking the plus button a bunch of times, it will look sorta like this:





Starting Point

First, create a new HTML document and ensure your starting point looks as follows:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Events</title>
  <script src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
  <style>
    #container {
      padding: 50px;
      background-color: #FFF;
    }
  </style>
</head>

<body>
  <div id="container"></div>
  <script type="text/babel">

  </script>
</body>

</html>
```

Once your new HTML document looks like what you see above, it's time to add our partially implemented counter example. Inside our script tag below the **container** div, add the following:

```
class Counter extends React.Component {
  render() {
    var textStyle = {
      fontSize: 72,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold"
    };

    return (
      <div style={textStyle}>
        {this.props.display}
      </div>
    );
  }
}

class CounterParent extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0
    };
  }
  render() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };

    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold",
      lineHeight: "3px"
    };

    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count} />
        <button style={buttonStyle}>+</button>
      </div>
    );
  }
}

ReactDOM.render(
  <div>
    <CounterParent />
  </div>,
  document.querySelector("#container")
);
```

Making the Button Click Do Something

Each time we click on the plus button, we want the value of our counter to increase by one. What we need to do is going to roughly look like this:

1.Listen for the click event on the button.
2.Implement the event handler where we react to the click and increase the value of our this.state.count property that our counter relies on.

We'll just go straight down the list...starting with listening for the click event. In React, you listen to an event by specifying everything inline in your JSX itself. More specifically, **you specify both the event you are listening for and the event handler that will get called all inside your markup**. To do this, find the return function inside our CounterParent component, and make the following highlighted change:

```
 .
  .
  .
return (
  <div style={backgroundStyle}>
    <Counter display={this.state.count}/>
    <button onClick={this.increase} style={buttonStyle}>+</button>
  </div>
);
```

What we've done is told React to call the increase function when the **onClick** event is overheard. Next, let's go ahead and implement the increase function - aka our event handler. Inside our CounterParent component, add the following highlighted lines:

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

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);
  }

  increase(e) {
    this.setState({
      count: this.state.count + 1
    });
  }
```

```
render() {
    var backgroundStyle = {
      padding: 50,
      backgroundColor: "#FFC53A",
      width: 250,
      height: 100,
      borderRadius: 10,
      textAlign: "center"
    };

    var buttonStyle = {
      fontSize: "1em",
      width: 30,
      height: 30,
      fontFamily: "sans-serif",
      color: "#333",
      fontWeight: "bold",
      lineHeight: "3px"
    };

    return (
      <div style={backgroundStyle}>
        <Counter display={this.state.count} />
        <button onClick={this.increase}
style={buttonStyle}>+</button>
      </div>
    );
  }
}
```

Event Properties

- As you know, our events pass what are known as event arguments to our event handler. These event arguments contain a bunch of properties that are specific to the type of event you are dealing with.

- In the regular DOM world, each event has its own type. For example, if you are dealing with a mouse event, your event and its event arguments object will be of type **MouseEvent**. This **MouseEvent** object will allow you to access mouse-specific information like which button was pressed or the screen position of the mouse click.

- Event arguments for a keyboard-related event are of type **KeyboardEvent**. Your **KeyboardEvent** object contains properties which (among many other things) allow you to figure out which key was actually pressed.

- Each Event type contains its own set of properties that you can access via the event handler for that event!

Meet Synthetic Events

In React, when you specify an event in JSX like we did with **onClick**, you are not directly dealing with regular DOM events. Instead, you are dealing with a React-specific event type known as a **SyntheticEvent**. Your event handlers don't get native event arguments of type **MouseEvent, KeyboardEvent,** etc. They always get event arguments of type **SyntheticEvent** that wrap your browser's native event instead.

Each SyntheticEvent contains the following properties:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

These properties should seem pretty straightforward...and generic! The non-generic stuff depends on what type of native event our **SyntheticEvent** is wrapping. This means that a **SyntheticEvent** that wraps a **MouseEvent** will have access to mouse-specific properties such as the following:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Similarly, a SyntheticEvent that wraps a KeyboardEvent will have access to these additional keyboard-related properties:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

- **Don't refer to traditional DOM event documentation when using Synthetic events and their properties.**
- Because the SyntheticEvent wraps your native DOM event, events and their properties may not map one-to-one. Some DOM events don't even exist in React.
- To avoid running into any issues, if you want to know the name of a Synthetic event or any of its properties, **refer to the React Event System document** instead.

Doing Stuff With Event Properties

- Right now, our counter example increments by one each time you click on the plus button. What we want to do is **increment our counter by ten when the Shift key on the keyboard is pressed** while clicking the plus button with our mouse.
- The way we are going to do that is by using the shiftKey property that exists on the SyntheticEvent when using the mouse:
- The way this property works is simple. If the Shift key is pressed when this mouse event fires, then the shiftKey property value is **true**. Otherwise, the shiftKey property value is **false**.
- To increment our counter by 10 when the Shift key is pressed, go back to our increase function and make the following highlighted changes:

```
increase(e) {
  var currentCount = this.state.count;

  if (e.shiftKey) {
    currentCount += 10;
  } else {
    currentCount += 1;
  }

  this.setState({
    count: currentCount
  });
}
```

- Once you've made the changes, preview our example in the browser. Each time you click on the plus button, your counter will increment by one just like it had always done. If you click on the plus button with your Shift key pressed, notice that our counter increments by 10 instead.

- The reason that all of this works is because we change our incrementing behavior depending on whether the Shift key is pressed or not. That is primarily handled by the following lines:

```
if (e.shiftKey) {
  currentCount += 10;
} else {
  currentCount += 1;
}
```

- If the `shiftKey` property on our `SyntheticEvent` event argument is **true**, we increment our counter by 10. If the `shiftKey` value is **false**, we just increment by 1.

More Eventing Shenanigans

You Can't Directly Listen to Events on Components

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

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);
  }

  increase(e) {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <PlusButton onClick={this.increase} />
      </div>
    );
  }
}
```

- On the surface, this line of JSX looks totally valid. Whensomebody clicks on our PlusButton component, the increase function will get called. In case you are curious, this is what our PlusButton component looks like:

```
class PlusButton extends React.Component {
  render() {
    return (
      <button>
        +
      </button>
    );
  }
}
```

Our PlusButton component doesn't do anything crazy. It only returns a single HTML element!

- There is no clear answer to any of those questions. It's too harsh to say that the solution is to simply not listen to events on components either. Fortunately, there is a workaround where we treat the event handler as a prop and pass it on to the component.
- Inside the component, we can then assign the event to a DOM element and set the event handler to the the value of the prop we just passed in. I realize that probably makes no sense, so let's walk through an example.
- Take a look on highlighted green color text:

```
class CounterParent extends React.Component {
  .
  .
  .
  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <PlusButton clickHandler={this.increase} />
      </div>
    );
  }
}
```

- In this example, we create a property called clickHandler whose value is the increase event handler. Inside our PlusButton component, we can then do something like this:

```
class PlusButton extends React.Component {
  render() {
    return (
      <button onClick={this.props.clickHandler}>
        +
      </button>
    );
  }
}
```

- On our button element, we specify the **onClick** event and set its value to the clickHandler prop. At runtime, this prop gets evaluated as our increase function, and clicking the plus button ensures the increase function gets called

Listening to Regular DOM Events

- If you thought the previous section was a doozy, wait till you see what we have here. Not all DOM events have SyntheticEvent equivalents. It may seem like you can just add the **on** prefix and capitalize the event you are listening for when specifying it inline in your JSX:

```
class Something extends React.Component {
  .
  .
  .
  handleMyEvent(e) {
    // do something
  }

  render() {
    return (
      <div myWeirdEvent={this.handleMyEvent}>Hello!</div>
    );
  }
}
```

- It doesn't work that way! For those events that aren't officially recognized by React, you have to use the traditional approach that uses **addEventListener** with a few extra hoops to jump through.

```
class Something extends React.Component {
        .
        .
        .
    handleMyEvent(e) {
        // do something
    }

    componentDidMount() {
        window.addEventListener("someEvent",
this.handleMyEvent);
    }

    componentWillUnmount() {
        window.removeEventListener("someEvent",
this.handleMyEvent);
    }

    render() {
        return (
            <div>Hello!</div>
        );
    }
}
```

- We have our Something component that listens for an event called **someEvent**.
- We start listening for this event under the **componentDidMount** method which is automatically called when our component gets rendered.
- our event is by using **addEventListener** and specifying both the event and the event handler to call.
- To removing the event listener when the component is about to be destroyed.
- To do that, you can use the opposite of the **componentDidMount** method, the **componentWillUnmount** method.
- Inside that method, put your **removeEventListener** call there to ensure no trace of our event listening takes place after our component goes away.

The Meaning of this Inside the Event Handler

- When dealing with events in React, the value of this inside your event handler is different than what you would normally see in the non-React DOM world. In the non-React world, the value of this inside an event handler refers to the element that fired the event:

```
function doSomething(e) {
  console.log(this); //button element
}

var foo = document.querySelector("button");
foo.addEventListener("click", doSomething, false);
```

- In the React world, the value of this does not refer to the element that fired the event. The value is the very unhelpful (yet correct) **undefined**. That is why we need to explicitly specify what this binds to using the bind method like we've seen a few times:

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

    this.state = {
      count: 0
    };

    this.increase = this.increase.bind(this);
  }

  increase(e) {
    console.log(this);

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

  render() {
    return (
      <div>
        <Counter display={this.state.count} />
        <button onClick={this.increase}>+</button>
      </div>
    );
  }
}
```

- In this example, the value of `this` inside the `increase` event handler refers to the **CounterParent** component.
- It doesn't refer to the element that triggered the event. You can attribute this behavior to us binding the value of `this` to our component from inside our **constructor.**

React...why? Why?!

- Why React decided to deviate from how we've worked with events in the past?

1.Browser Compatibility
2.Improved Performance

1. **Browser Compatibility**
- Event handling is one of those things that works consistently in modern browsers, but once you go back to older browser versions, things get really bad really quickly.
- By wrapping all of the native events as an object of type SyntheticEvent, React frees you from dealing with event handling quirks that you will end up having to deal with otherwise.

2. **Improved Performance**
- In complex UIs, the more event handlers you have, the more memory your app takes up.
- React never attaches event handlers to the DOM elements directly. **It uses one event handler at the root of your document** that is responsible for listening to all events and calling the appropriate event handler as necessary:

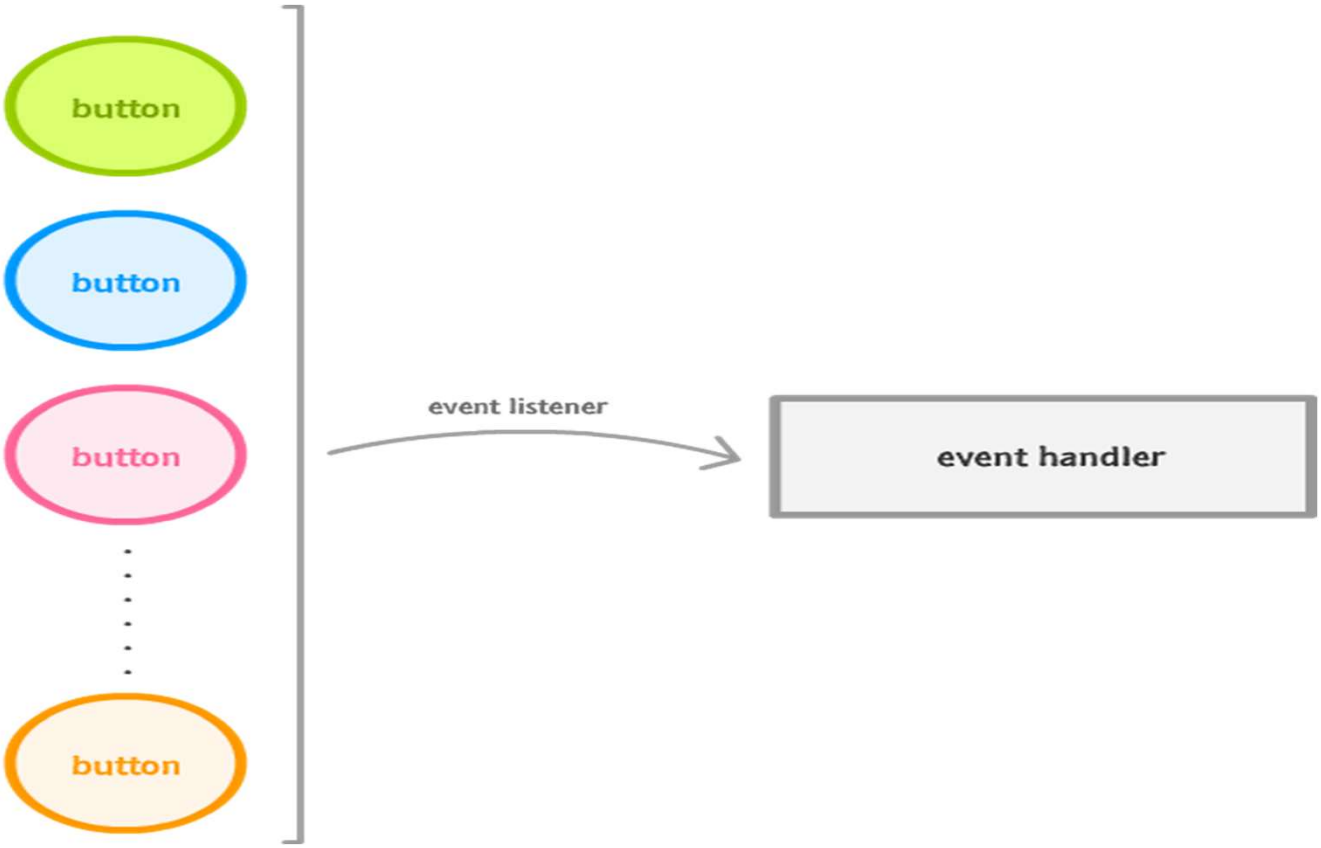


Fig: React uses one event handler at the root of your document.

Conclusion

- You'll spend a lot of time dealing with events, and this tutorial threw a lot of things at you.
- We started by learning the basics of how to listen to events and specify the event handler.
- Towards the end, we were all the way in the deep end and looking at eventing corner cases that you will bump into if you aren't careful enough.

Thank You