React Events

Goals

- Attach event handlers to components in React
- Use method binding to preserve the ***this*** context with event handlers
- Pass event handlers down as props to child components
- Understand the ***key*** prop that React asks for when mapping over data

React Events Review

React Events

You can attach event handlers to HTML elements in React via special reserved attributes.

(You can do this in vanilla JS too, though the syntax is a bit different.)

Event Attributes

Any event you can listen for in JS, you can listen for in React.

Examples:

- Mouse events: `onClick`, `onMouseOver`, etc
- Form events: `onSubmit`, etc
- Keyboard events: `onKeyDown`, `onKeyUp`, `onKeyPress`
- Full list <https://reactjs.org/docs/events.html#supported-events>

Example

demo/events-examples/src/WiseSquare.js

```
import React, { Component } from "react";
import "./WiseSquare.css";

class WiseSquare extends Component {
  dispenseWisdom() {
    let messages = [ /* wise messages go here */
    ];
    let rIndex = Math.floor(Math.random() * messages.length);
    console.log(messages[rIndex]);
  }

  render() {
    return (
      <div className="WiseSquare"
           onMouseEnter={this.dispenseWisdom}>
```

```
        😀
      </div>
    );
  }
}

export default WiseSquare;
```

Method Binding

The keyword *this*

- When your event handlers reference the keyword ***this***, watch out!
- You will lose the ***this*** context when you pass a function as a handler
- Let's see what happens when we try to move our quotes into ***defaultProps***

Example Revisited

demo/events-examples/src/WiseSquareWithProps.js

```
class WiseSquareWithProps extends Component {
  static defaultProps = {
    messages: [ /* wise messages go here */
    ]
  }

  dispenseWisdom() {
    console.log("THIS IS:", this) // undefined 😭
    let { messages } = this.props;
    let rIndex = Math.floor(Math.random() * messages.length);
    console.log(messages[rIndex]);
  }

  render() {
    return (
      <div className="WiseSquare"
           onMouseEnter={this.dispenseWisdom}>
        😀
      </div>
    );
  }
}
```

Fixing our binding

There are three ways to fix this:

1. Use ***bind*** inline
2. Use an arrow function
3. Method bind in the constructor

Inline

```
<div className="WiseSquare"
      onMouseEnter={this.dispenseWisdom.bind(this)} >
  {/* */}
</div>
```

Pros

- Very Explicit

Cons

- What if you need to pass `this.dispenseWisdom` to multiple components?
- new function created on every render

Arrow Functions

```
<div className="WiseSquare"
      onMouseEnter={() => this.dispenseWisdom()} >
  {/* */}
</div>
```

Pros

- No mention of bind!

Cons

- Intent less clear
- Again, what if you need to pass the fn to multiple components?
- new function created on every render

In the constructor

```
class WiseSquareWithProps extends Component {
  constructor(props) {
    super(props);
    /* do other stuff */
    this.dispenseWisdom = this.dispenseWisdom.bind(this);
  }
}
```

Pros

- Only need to bind once!
- More performant

Cons

- Hot reloading won't apply

Method Binding with Arguments

In our previous examples, `this.dispenseWisdom` didn't take any arguments.

But what if we need to pass arguments to an event handler?

An Example

demo/events-examples/src/ButtonList.js

```
class ButtonList extends Component {
  static defaultProps = {
    colors: ["green", "red", "blue", "peachpuff"]
  };

  handleClick(color) {
    console.log(`You clicked on the ${color} button.`);
  }

  render() {
    return (
      <div className="ButtonList">
        {this.props.colors.map(c => {
          const colorObj = { backgroundColor: c };
          return (
            <button style={colorObj}
                    onClick={this.handleClick.bind(this, c)}>
               Click on me!</button>
          );
        })}
      </div>
    );
  }
}
```

- Inside of a loop, you can bind and pass in additional arguments
- Also possible to use an arrow function
- Both these approaches suffer from the same performance downsides we've already seen
- We can do better, but first we need to talk about…

Passing functions to child components

- A very common pattern in React
- The idea: children are often not stateful, but need to tell parents to change state
- How we send data "back up" to a parent component

How data flows

- A parent component defines a function
- The function is passed as a prop to a child component
- The child component invokes the prop
- The parent function is called, usually setting new state
- The parent component is re-rendered along with its children

What it looks like

demo/numbers-app/src/NumberList.js

```
class NumberList extends Component {
  constructor(props) {
    super(props);
    this.state = { nums: [1, 2, 3, 4, 5] };
  }

  remove(num) {
    this.setState(st => ({
      nums: st.nums.filter(n => n !== num)
    }));
  }

  render() {
    let nums = this.state.nums.map(n => (
      <NumberItem value={n} remove={() => this.remove(n)} />
    ));
    return <ul>{nums}</ul>;
  }
}
```

demo/numbers-app/src/NumberItem.js

```
class NumberItem extends Component {
  render(){
    return(
      <li>
        {this.props.value}
        <button
          onClick={this.props.remove}>
          X
        </button>
      </li>
    )
  }
}
```

- We could also method bind inside of the ***map***
- In fact, we can do even better!

Using a single bound function

demo/numbers-app/src/BetterNumList.js

```
class BetterNumList extends Component {
  constructor(props) {
    super(props);
    this.state = { nums: [1, 2, 3, 4, 5] };
    this.remove = this.remove.bind(this);
  }

  remove(num) {
    this.setState(st => ({
      nums: st.nums.filter(n => n !== num)}))
  }

  render() {
    let nums = this.state.nums.map(n => (
      <BetterNumItem value={n}
        remove={this.remove} />
    ));
    return <ul>{nums}</ul>;
  }
}
```

demo/numbers-app/src/BetterNumItem.js

```
class NumberItem extends Component {
  constructor(props) {
    super(props);
    this.handleRemove =
      this.handleRemove.bind(this);
  }

  handleRemove() {
    this.props.remove(this.props.value);
  }

  render(){
    return(
      <li>
        {this.props.value}
        <button
          onClick={this.handleRemove}>
          X
        </button>
      </li>
    )
  }
}
```

Where to bind?

- The higher the better - don't bind in the child component if not needed.
- If you need a parameter, pass it down to the child as a prop, then bind in parent and child
- Avoid inline arrow functions / binding if possible

- No need to bind in the constructor **and** make an inline function
- If you get stuck, don't worry about performance, just try to get the communication working
  - You can always refactor later!

Naming Conventions

- You can call these handlers whatever you want - React doesn't care
- For consistency, try to follow the `action` / `handleAction` pattern:
  - In the parent, give the function a name corresponding to the behavior (`remove`, `add`, `open`, `toggle`, etc.)
  - In the child, use the name of the action along with "handle" to name the event handler (`handleRemove`, `handleAdd`, `handleOpen`, `handleToggle`, etc.)

Lists and Keys

demo/numbers-app/src/BetterNumList.js

```
class BetterNumList extends Component {
  render() {
    let nums = this.state.nums.map(n => (
      <BetterNumItem value={n}
        remove={this.remove} />
    ));
    return <ul>{nums}</ul>;
  }
}
```

- When mapping over data and returning components, you get a warning about keys for list items
- ***key*** is a special string attr to include when creating lists of elements

Adding keys

Let's assign a key to our list items inside ***numbers.map()***

```
class NumberList extends Component {
  render() {
    const nums = this.state.nums.map(n => (
      <NumberItem value={n}
        key={n}
        remove={this.remove}
      />
    ));
    return <ul>{nums}</ul>;
  }
}
```

Keys

- Keys help React identify which items are changed/added/removed.
- Keys should be given to repeated elems to provide a stable identity.

Picking a key

- Best way: use string that uniquely identifies item among siblings.
- Most often you would use IDs from your data as keys:

```
let todoItems = this.state.todos.map(todo =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Last resort

When you don’t have stable IDs for rendered items, you may use the iteration index as a key as a last resort:

```
// Only do this if items have no stable IDs

const todoItems = this.state.todos.map((todo, index) =>
  <li key={index}>
    {todo.text}
  </li>
);
```

- Don’t use indexes for keys if item order may change or items can be deleted.
  - This can cause performance problems or bugs with component state.