Building Web Applications with React

Chapter 2: State and Events

Chapter Objectives

In this chapter, we will:

- Introduce `props`
- Manage component state
- Utilize React events in components
- Leverage lifecycle methods and hooks

Chapter Concepts

Unidirectional Data Flow

State and Events

Hooks

Lifecycle Events

Chapter Summary

Unidirectional Data Flow

- React components are purely UI objects
  - Render elements based on current data
  - Trigger events in response to user actions
    - More on React events later in this chapter
- Data should flow in one direction only
  - From parent to child component
  - Reduces complexity
  - Enhances maintainability
- Where does the parent component receive its data?
  - React is not prescriptive
    - Components can retrieve their own data
      - Not the preferred pattern
  - Facebook recommends the Flux pattern
    - Discussed in a later chapter

Component Properties

- Parent components pass properties to children
  - In this context, the parent is also known as the *owner*
  - Later, we will see pure parent components known as *containers*
    - Containers contain only logic and no JSX
- Properties are immutable
  - The child component cannot change them
- Passed to the child component as attributes of the tag

```
<Book author={book.author} title={book.title} />
```

Receiving Component Properties

- In class components, properties are available as `this.props`

```
render() {
   return (<tr>
             <td> {this.props.title}</td>
             <td> {this.props.author}</td>
           </tr>);
 }
```

- In functional components, props are passed in as an argument

```
function Book(props) {
  const { title, author } = props;
  return (
      <tr><td> {title} </td>
          <td>{author}</td></tr>
  )a
}
```

Destructure `props` to access individual properties passed in to component

Defining Properties

- Best practice for child component to specify its expected properties
  - `Component.propTypes` object literal defines:
    - Type
    - Name
    - And whether property required
- Missing or incorrect property generates compiler warning
  - Only checked in development mode for performance reasons
  - Does not cause runtime error
    - Although absence or incompatible property may cause other errors

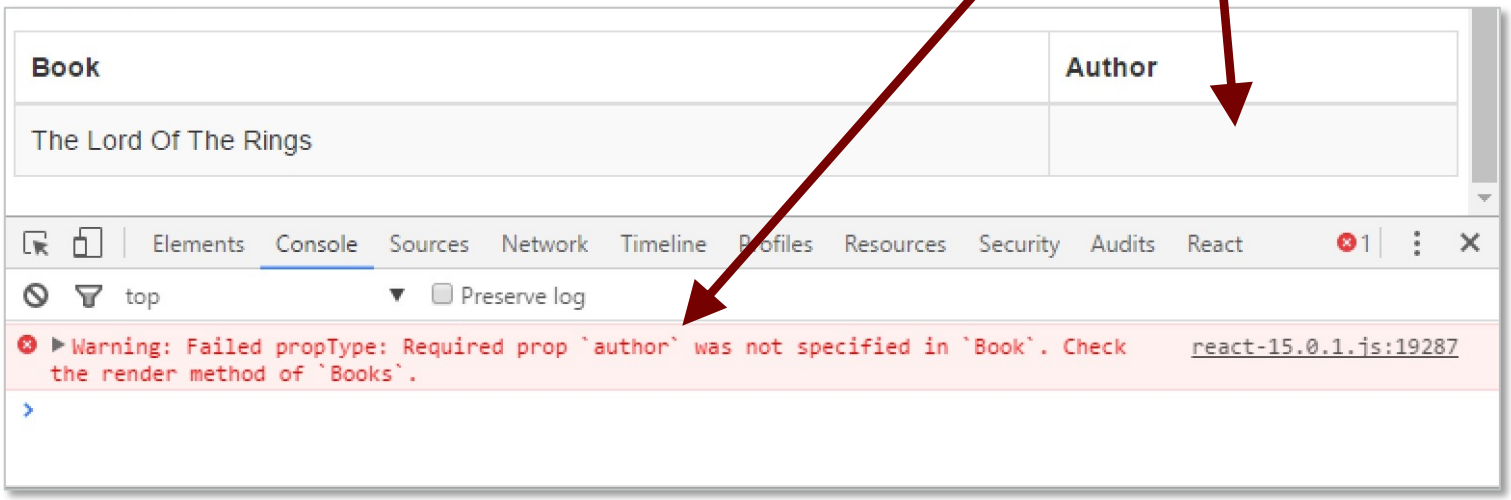
```
class Book extends React.Component {
    render() {
        return (<tr><td>{this.props.title}</td>
            <td>{this.props.author}</td></tr>);
}  }

Book.propTypes = {
    title: PropTypes.string.isRequired,
    author: PropTypes.string.isRequired
};
```

PropTypes

- `PropTypes` is defined in `prop-types` npm package
  - Must be imported before it can be used
- `PropTypes` validators include:
  - `PropTypes.array`
  - `PropTypes.bool`
  - `PropTypes.func`
  - `PropTypes.number`
  - `PropTypes.object`
  - `PropTypes.string`
- By default, all properties are optional
  - Specify `isRequired` for compulsory properties
    - `PropTypes.func.isRequired`
  - Causes compiler warning, not runtime error

Missing `PropTypes` Illustrated

```
Book.propTypes = {title: PropTypes.string.isRequired,
                  author: PropTypes.string.isRequired};
```



Defining Default Property Values

- Sometimes useful to define default values for properties
  - Allow for null or missing entries in underlying data store
  - Provide suitable alternative placeholder data
- Different approaches for class and functional components
- Functional components use JavaScript default argument values

```
const { title = "unknown", author  = "unknown" } = props;
```

- Class-based components use `defaultProps`
  - Specified as an object-literal property of the component
    - Add named property value pairs matching component props
- React will invoke `getDefaultProps()` lifecycle method only once
  - As component is created
- Merges `defaultProps` with props specified by parent component
  - Parent component values take precedence
  - `propTypes.required` attribute is no longer needed

`defaultProps` Code Sample

```
Book.defaultProps = {
        title: 'unknown',
        author: 'unknown'
};
```

`defaultProps` added to `Book` component

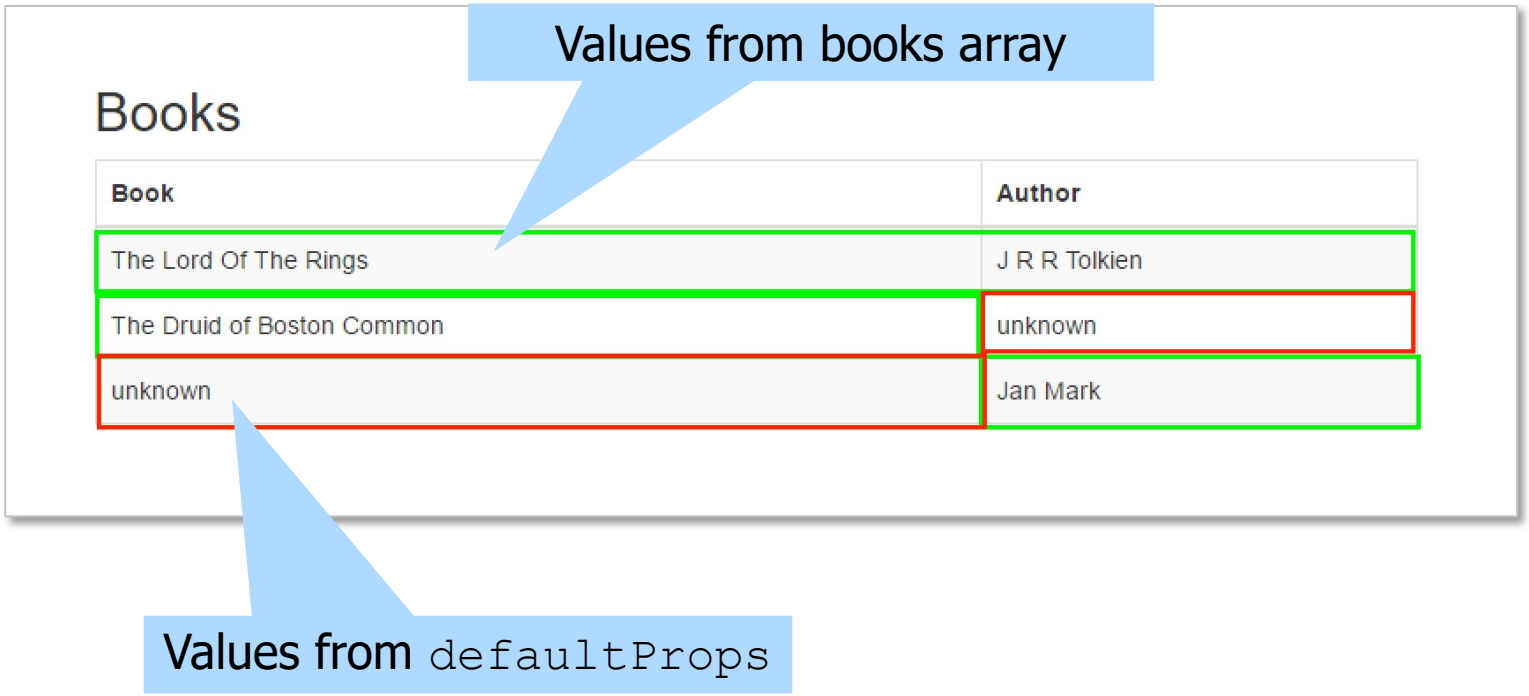
```
const books = [{
    title: "The Lord Of The Rings",
    author: "J R Tolkien" },
   {
    title: "The Druid of Boston Common" },
   {
    author: "Jan Mark"
   }];
```

Missing properties in books array

```
{books.map(function (item, i) {
  return <Book author={item.author}
               title={item.title} key={i} />;
})}
```

Some values not specified in books array

`defaultProps` Illustrated



JSX Spread Attributes

- Allow `props` to be defined as an object
  - Object can be expanded into component as `props`
    - Uses ES6 spread operator `...`

```
let props = {};

props.author = "Diana Wynne Jones";
props.title = "Archer's Goon";

const component = <Book {...props} />;
```

Components can be stored in variables and inserted into JSX

Spread operator expands object properties as props

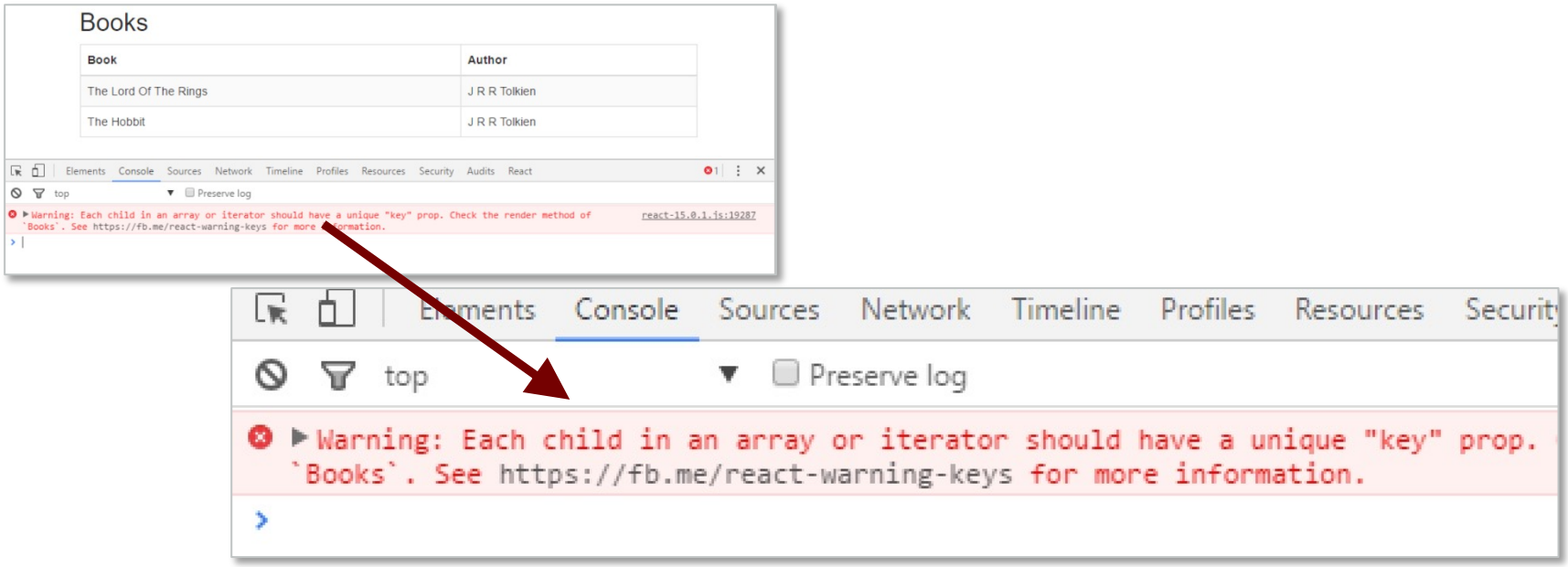
Iterating Through Array Properties

- Inside JSX, use `map()` to iterate through arrays

```
{books.map(function (item) {
    return (<Book author={item.author} title={item.title} />);
})}
```

The Missing Key

- The code on the previous slide has a problem
  - No key was supplied to help React track dynamic children



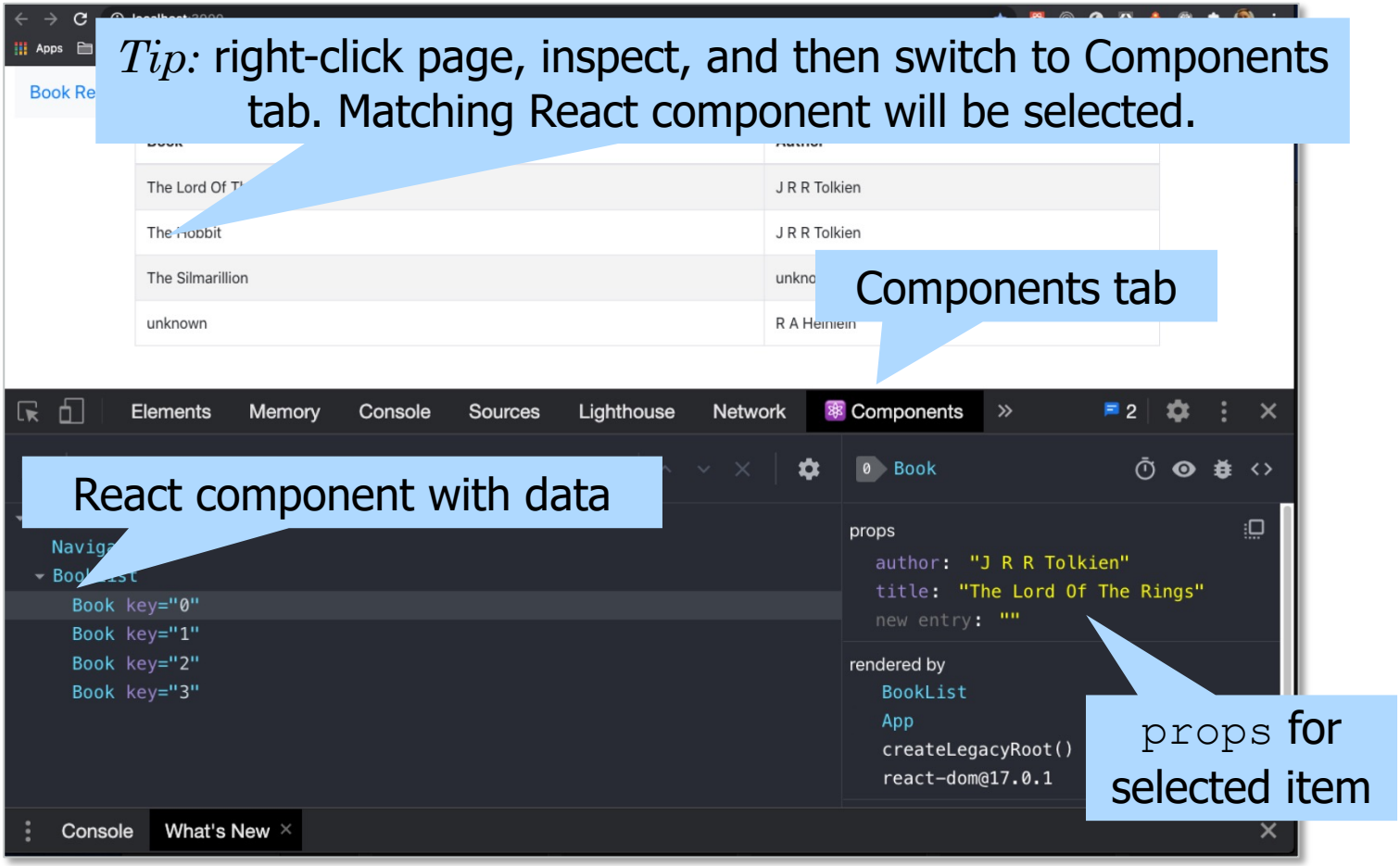
Dynamic Child Reconciliation

- Reconciliation: the process React uses to update the DOM on each render
  - Keeps track of child order for efficient updates
    - Can become very complicated with dynamic children
      - Especially if sorted after creation
- Developer should always supply a unique key
  - Ensures efficient tracking of dynamic children

```
{books.map(function (item, i) {
  return (<Book  author={item.author}
                 title={item.title}
                 key={i} />);
})}
```

Demo: React Developer Tools

- Your instructor will demonstrate the React Developer Tools extension



Exercise 2.1: Passing Properties to Components

- In this exercise, you will set up your `Book` component to receive `props`, and pass the values in from the `BookList` parent component
- Please refer to the Exercise Manual

Chapter Concepts

Unidirectional Data Flow

State and Events

Hooks

Lifecycle Events

Chapter Summary

Managing State

- React thinks of components as state machines
  - Components simply render based on their current state
    - Removing need for complex updates
- `Props` provide immutable state
- Sometimes state needs to change
  - Based on user actions or computation
- Different approaches for functional and class-based components
  - Same underlying behavior
  - Very different syntax
- We will cover class-based components first

The State Property

- Mutable component state in class components is available as `this.state`
- Initialized in the constructor when using ES6 syntax
  - In earlier versions of React, set via `setInitialState()`
    - A component lifecycle method
      - More on lifecycle methods later

Initializing State

```
class Books extends React.Component {

    constructor() {
        super();
        this.state = {
            books: [{ title: "The Lord Of The Rings",
                      author: "J R R Tolkien" },
            {
                title: "The Hobbit",
                author: "J R R Tolkien"
            }]
        };
    }

    render() {
```

Adding `books` array to the state object in the constructor

Accessing `books` variable on `this.state`

```
{this.state.books.map(function (item, i) {
    return <Book author={item.author}
                 title={item.title} key={i} />;
})}
```

Setting State

- State should only be assigned once, in the constructor
- Subsequent state updates should use the `setState()` method
  - Expects an object literal
  - Specify the property to update and the value to assign
- Calling `setState()` will trigger the component to re-render

```
constructor() {
        super();
        this.state = {title: "Add Book"};
}

setTitle(e) {
        this.setState({title: e.target.value});
}
```

`e.target.value` contains value of form element that triggered event

Controlled vs. Uncontrolled Inputs

- Common to trigger state change based on user input
- Two kinds of input controls in React
  - Uncontrolled inputs
  - Controlled inputs
- Uncontrolled inputs can be used to call methods that set the state
  - Do not get their value from state
- Controlled inputs have their value set from state
  - Also update state in response to user actions
- The component will re-render whenever either input type updates state

Inputs Illustrated

Uncontrolled input has no value specified

```
<input type="text" onChange={this.setTitle.bind(this)} />
```

Controlled input both assigns value and calls method to update state

```
<input type="text" value={this.state.title}
                   onChange={this.setTitle.bind(this)} />
```

- Both inputs use `bind(this)` to set context of `this` to the component
  - Ensures `this.setState()` is available inside the method

Alternative Binding Syntax

```
<input type="text" value={this.state.title}
       onChange={e => this.setTitle(e)}  />
```

Arrow functions automatically preserve context of `this`

Updating Stateful Data

- The controlled input on the previous slide would work for data inserts
  - Object could be created and passed to data store
- Problem: what if the data already exists?
- Data is passed from parent to child components as `props`
  - Props are immutable
- The following input would be read-only
  - Any changes the user made would trigger a re-render
  - Re-rendered input would display the original prop

```
<input type="text" value={this.props.title}
                   onChange={this.setTitle.bind(this)} />
```

- Solution: pass the update function as a prop from the parent component
  - Parent component has access to underlying state
    - We will see alternative solutions later when we consider Flux

Controlled Inputs with Props

Note that `this` is bound to the context of the *parent* component

```
setTitle(e) {
 this.setState({title: e.target.value});
}

render() {
 return (<div><UpdateBookForm change={this.setTitle.bind(this)}
                              title={this.state.title} /> </div>);
}
```

The child component requires a `func` prop

```
child.propTypes = {change: React.PropTypes.func.isRequired,
                   title: React.PropTypes.string.isRequired };
```

Child component does not set context of `this`

```
<input type="text" value={this.props.title}
                          onChange={this.props.change} />
```

Demo: Controlled vs. Uncontrolled

- Your instructor will now demonstrate the difference between controlled and uncontrolled components

The React Event System

- React events look like standard DOM events
  - `onClick`
  - `onChange`
  - `onSubmit`
- Implementation is entirely different
- Event handlers are passed an instance of `SyntheticEvent`
  - Behavior emulates native events
    - `e.stopPropagation()`
    - `e.preventDefault()`
  - Has two key advantages
    - Events work identically across all browsers
    - Events are pooled for efficiency and performance

Event Binding

- Event binding in React changed with the introduction of ES6 syntax
- In pre-ES6 syntax, events were automatically bound
  - No need to specify the context for `this`
- In the ES6 syntax, developer specifies binding
  - If necessary, bind `this` to parent or child as appropriate
    - Additional arguments can also be bound

```
<input type="text" onChange={this.setTitle.bind(this)} />
```

- Alternatively, can use ES6 arrow syntax to preserve context of this

```
<input type="text" onChange={e => this.setTitle(e)}   />
```

`e` is the value of the input

Exercise 2.2: Working with Forms, State, and Events

- In this exercise, you will create a React form and update `this.state` using events in a class-based component
- Please refer to the Exercise Manual

Chapter Concepts

Unidirectional Data Flow

State and Events

Hooks

Lifecycle Events

Chapter Summary

State in Functional Components

- Functional components were created to simplify React
  - Originally for *very* simple functions that just returned JSX
  - Can now be used in complex scenarios thanks to *hooks*
- Hooks allow functions to 'hook into' React state and lifecycle events
  - More on lifecycle events soon
- The `useState` hook simplifies state management
  - Replaces `setState` in class components
  - Is available as a non-default export in React

The `useState` Hook

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

`useState` is one of the most frequently used hooks

```
const [content, setContent] = useState("");
```

`useState` returns an array. First value is the variable whose state is stored. The second is the setter method that updates the variable.

```
<input type="text" className="form-control"
       value={content}
       onChange={(e) => setContent(e.target.value)} />
```

Call the setter method inside your code

Hooks

- Hooks give access to behavior previously only available in classes
  - Allowing functions to become first-class components
- Hooks solve problems of complexity caused by classes
  - Make it easier to decompose components into smaller functions
    - Easier to test and maintain
  - Classes can be harder to optimize during the build process
- Underlying concepts are the same as when using classes
  - Intention is to reduce the complexity of implementation
- Can write custom hooks to extract component logic into reusable functions
  - Allows new way to share behavior and stateful logic
    - Without needing to modify component hierarchy
  - You will create a custom hook later in this course

Built-in Hooks

- React comes with several hooks
  - Developers also write their own additional hooks
- Hooks do not have a fixed signature either for arguments or return
  - Some hooks accept no arguments, some accept several
  - Some hooks return simple values, others return complex arrays
    - Arrays might contain a mix of data and functions
      - As in the `useState` hook
- Two very frequently used hooks
  - `useState`
    - Manages local state
  - `useEffect`
    - Replaces many lifecycle methods from class components
      - More on lifecycle methods soon

Some Additional Hooks

- Other built-in hooks are available, including:
  - `useContext`
    - Gives access to global React state
    - Avoids having to pass props down lengthy component trees
  - `useReducer`
    - Alternative to `useState` using the Redux pattern without Redux
      - More on Redux later
  - `useRef`
    - Holds a mutable value in `ref.current`
    - Can hold a DOM reference
      - Though `useCallback` more suitable in some circumstances
- Same hook can be used repeatedly inside the component

```
const [book, setBook] = useState('The Hobbit');
const [author, setAuthor] = useState('J R R Tolkien');
const [review, setReview] = useState('Nonesuch');
```

Exercise 2.3: Using State in Functional Components

- In this exercise, you will add state to functional components with the `useState` hook
- Please refer to the Exercise Manual

Chapter Concepts

Unidirectional Data Flow

State and Events

Hooks

Lifecycle Events

Chapter Summary

Debrief: Rendering Overload

- Follow along as your instructor opens the exercise solution in Google Chrome, and:
  - Opens the Developer Tools
  - Navigates to the source file for `Book.jsx`
  - Sets a breakpoint in the `render()` method
  - Uses the form to add a new book
- Note that the `render()` method is being called for *every* book
  - Even if they are already part of the DOM
- This is inefficient
  - Though not quite as inefficient as it seems
    - The actual DOM is not being updated, just the virtual DOM
- Solution is to use the React lifecycle
  - Class components us lifecycle methods
  - Functional components use hooks and other alternatives

React Component Lifecycle Methods

- Two variations of the lifecycle
  - Events that run on initial creation of the component
  - Events that run as component changes after initial load
- `render()` is common to both
- Class components use lifecycle methods to respond to lifecycle events
  - Functional components use hooks
  - We'll consider lifecycle methods first
- `render()` is the only compulsory lifecycle method
  - Returns a single item
    - DOM node
    - Component
    - Component Tree
  - Does *not*:
    - Update state
    - Interact directly with the browser

Initial React Component Lifecycle

- `getDefaultProps()`
  - Sets the initial values of `this.props` to any `defaultProps`
  - (Replaced by constructor in ES6)
- `getInitialState()`
  - Sets the initial value of `this.state`
  - (Replaced by constructor in ES6)
- `componentWillMount()`
  - Set up non-UI event listeners
- `render()`
- `componentDidMount()`
  - DOM available
  - Set up timers
  - Integrate with third-party libraries
- `componentWillUnmount()`
  - Remove non-UI event listeners

Subsequent Lifecycle

- `componentWillReceiveProps()`
  - Opportunity to respond to props changing before render
- `shouldComponentUpdate()`
  - Has access to existing and changed `props` and `state`
  - Return `false` to prevent component from re-rendering
- `componentWillUpdate()`
  - Respond to updates before rendering
  - Cannot be used to call `setState()`
  - Considered unsafe and should no longer be used
- `render()`
- `componentDidUpdate`
  - Updates have been flushed to the DOM

shouldComponentUpdate()

- Determines whether the component needs to render
  - Compare current and new `props` and `state`
    - Return `true` if the component should render, `false` if not

Any new `props` and `state` passed to method as arguments

```
shouldComponentUpdate(nextProps, nextState) {
    if(this.state.book && nextState.book) {
        if(this.state.book.Id != nextState.book.Id){
            return true;
        }
        }
        if(this.props.approved && nextProps.approved)
        if(this.props.approved != nextProps.approved) {
        return true;
        }
    return false;
    }
```

Check that variables exist before attempting to access values

Hooking into the React Lifecycle

- Lifecycle methods help prevent side effects
  - The render method should not update state
    - Could cause continual re-rendering
- Lifecycle methods are not available inside functional components
  - The `useEffect` hook performs the same function
  - Runs *after* render
  - Combines behavior from several lifecycle events
    - `componentDidMount`
    - `componentDidUpdate`
    - `componentWillUnmount`
- Not every lifecycle method is replaced by a hook
  - Equivalent of `shouldComponentUpdate()` is `React.memo()`
  - Creates memoized version of return from `render()`
  - Shallowly compares `props`
  - If `props` have not changed, returns memoized version

The `useEffect` Hook

- Two ways to call `useEffect`
  - Providing a single function
    - Code will run after every render

```
useEffect(() => {
    console.log("I run after every render");
});
```

  - Providing an additional dependency array
    - Code will only run after render if a dependency has changed

```
useEffect(() => {
    console.log("I only run after render if bookId "
                + bookId + " changes");
    },
[bookId]);
```

Second argument is optional dependency array

Exercise 2.4: Improving Performance with `shouldComponentUpdate()`

- In this exercise, you will prevent unnecessary calls to the `render()` method with the `shouldComponentUpdate()` component lifecycle method and `React.memo`
- Please refer to the Exercise Manual

Chapter Concepts

Unidirectional Data Flow

State and Events

Hooks

Lifecycle Events

Chapter Summary

Chapter Summary

In this chapter, we have:

- Introduced `this.props`
- Managed component state
- Utilized React events in components
- Leveraged lifecycle methods and hooks