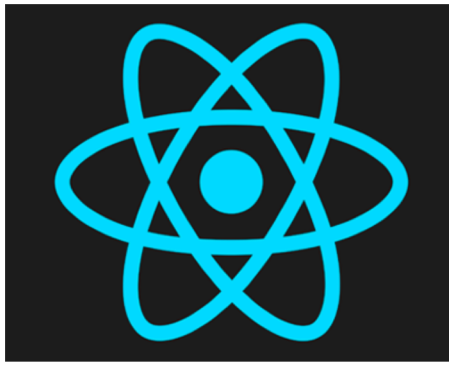
Module 1

React Overview

React, What is?

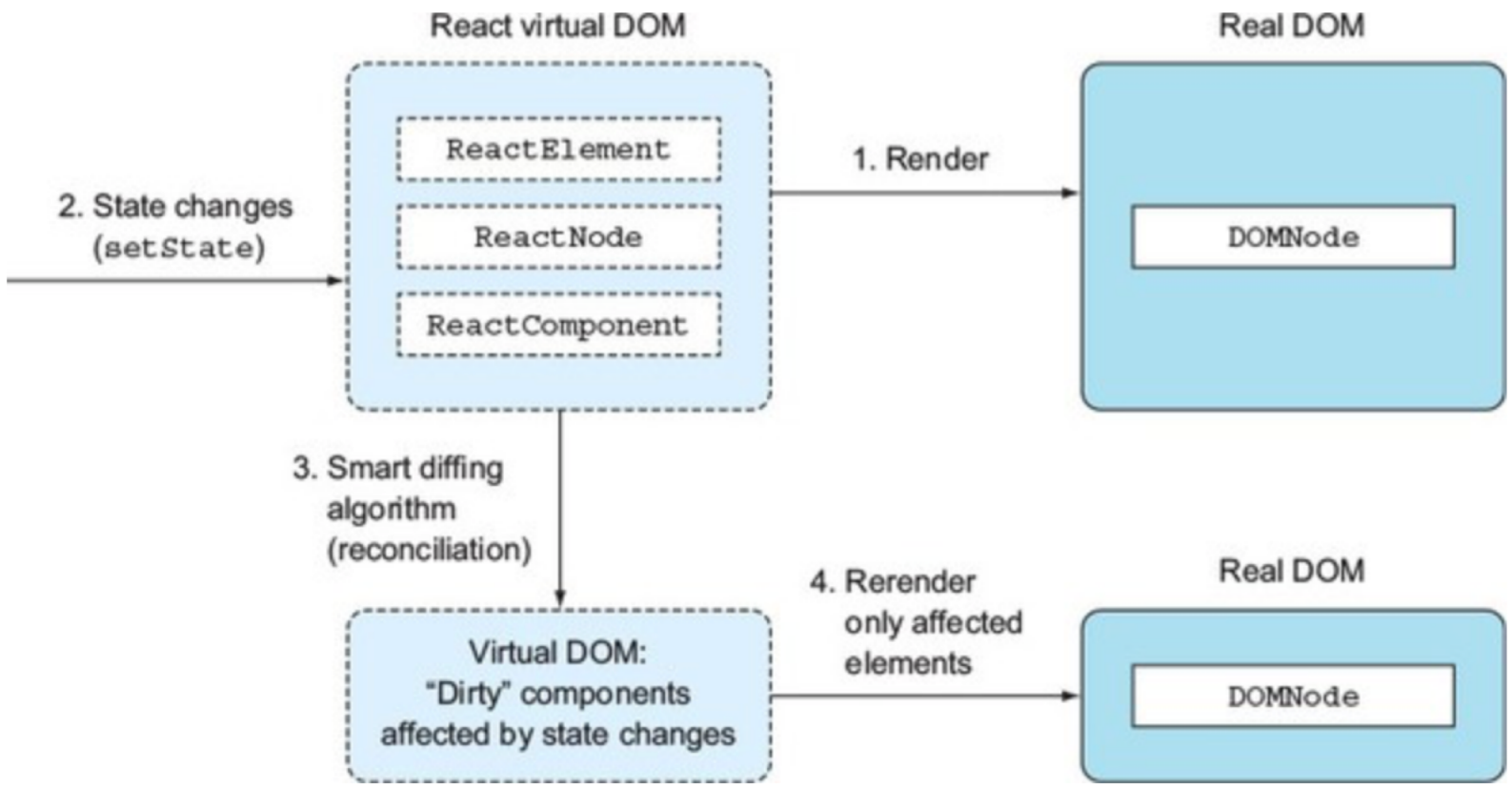
- It’s a UI component library
- The UI components are created with React using JavaScript
- React UI components are highly self-contained, concern-specific blocks of functionality
  - Such components have both a visual representation and dynamic logic
- React emerged from Facebook
- Benefits:
  - Simpler Apps – Component based with pure JS
  - Fast UI – Virtual DOM and reconciliation algorithm
  - Less code to write – Variety library and components
  - Render on server using Node and power mobile apps using React Native



React

- React uses a *virtual DOM* to find differences (the delta) between what's already in the browser and the new view
  - To allow React to create a Virtual DOM, you will need React's Components
  - To make any HTML code a static React Component, you just need to return HTML Code in React's **render() method**
  - You will need to replace class attribute name to **className** of HTML elements in render() method — because class is a reserved word in JavaScript
  - The React Component gets converted to the React Element
  - Whenever React renders React Elements, every single virtual DOM object gets update
  - Once the virtual DOM has updated, then React compares the virtual DOM with a virtual DOM snapshot that was taken before the update finds out *exactly which virtual DOM objects have changed (**diffing or reconciliation**)*

React virtual DOM



React simple component

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  document.getElementById('hello-example')
);
```

ReactDOM

- The react-dom package provides **DOM-specific methods** that can be used at the top level of your app
- React supports all popular browsers, including Internet Explorer 9 and above, although some polyfills are required for older browsers such as IE 9 and IE 10.

```
ReactDOM.render(element, container[, callback])
```

  - Render a React element into the DOM in the supplied container and return a reference to the component (or returns null for stateless components).

How to add React to a project

- Using CDN

```
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
```

  - The **last script allows to work with JSX**, Now you can use JSX in any <script> tag by adding type="text/babel"attribute to it.

- Create a React App
  - Best way to start building Single Page Applications
  - You'll need to have Node >= 6 and npm >= 5.2 on your machine.

```
npx create-react-app my-app
cd my-app
npm start
```

JSX

- it is a syntax extension to JavaScript
- React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code
- Embedding Expressions in JSX

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

JSX

- You can put any valid JavaScript expression inside the curly braces in JSX.
  - For example, 2 + 2
  - user.firstName
  - formatName(use

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

JSX

- After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.
- This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX

- Specifying attributes with JSX
  - You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>;
```

  - You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

  - Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

JSX

- Specifying Children with JSX

```
const element = <img src={user.avatarUrl} />;
```

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX

- JSX Prevents Injection Attacks
  - By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

JSX

- JSX Represents Objects
  - Babel compiles JSX down to React.createElement() calls.

```jsx
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

With JSX

```jsx
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Without JSX

JSX

```
<div>
  <HelloWorld/>
  <br/>
  <a href="http://webapplog.com">Great JS Resources</a>
</div>
```

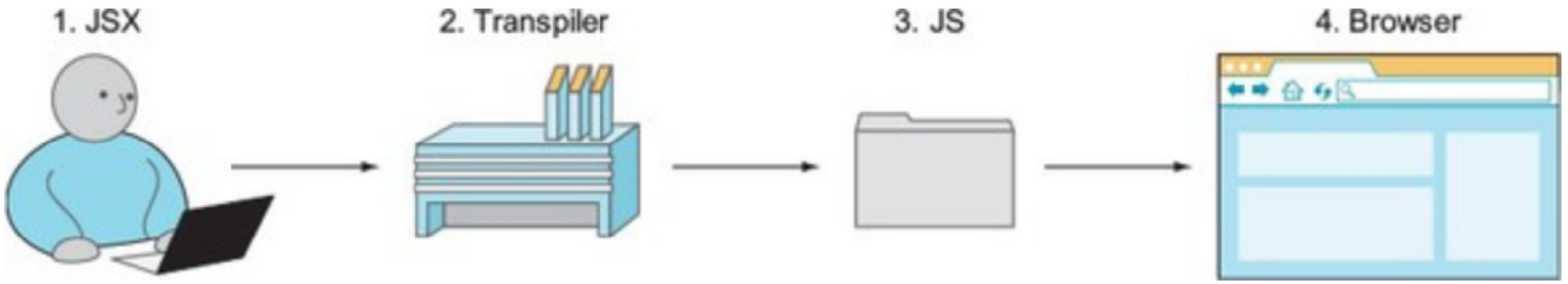
With JSX

```
React.createElement(
  "div",
  null,
  React.createElement(HelloWorld, null),
  React.createElement("br", null),
  React.createElement(
    "a",
    { href: "http://webapplog.com" },
    "Great JS Resources"
  )
)
```

Without JSX

JSX

- In essence, JSX is a small language with an XML-like syntax
- We build highly interactive UIs, and JS and HTML are tightly coupled to implement various pieces of functionality.
- React fixes the broken separation of concerns (SoC) principle by bringing together the description of the UI and the JS logic; and with JSX, the code looks like HTML and is easier to read and write



React considerations

- Developers need to pair it with a library like **Redux or React Router** to achieve functionality comparable to Angular or Ember
  - Redux is a predictable state container for JavaScript apps.
  - React Router will be the source of truth for your URL
- React's core API is still changing
- React only has a one-way binding
- **React isn't reactive** (as in reactive programming and architecture, which are more event-driven, resilient, and responsive) out of the box
  - You need Reactive extensions

Components in React

- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.
- You use ES6 class to define a component

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- the render function inside a component is also responsible for dealing with JSX.

Call a component

```
<div id="container"></div>

<script type="text/babel">
class HelloWorld extends React.Component {
    render(){
        return <p>Hello, I am a component!</p>
    }
}

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

React properties

- HTML elements can be customized by the attribute values contained in the opening tag.

```
class HelloWorld extends React.Component {
    render(){
        return <p>Hello, {this.props.greetTarget}!</p>
    }
}

ReactDOM.render(
    <div>
        <HelloWorld greetTarget="World"/>
        <HelloWorld greetTarget="my baby"/>
    </div>,document.querySelector("#container")
);
```

React properties

- **this.props.children**
  - It is used to display whatever you include between the opening and closing tags when **invoking** a component.

```
const Picture = (props) => {
  return (
    <div>
      <img src={props.src}/>
      {props.children}
    </div>
  )
}
```

```
//App.js

render () {
  return (
    <div className='container'>
      <Picture key={picture.id} src={picture.src}>
          //what is placed here is passed as props.children
      </Picture>
    </div>
  )
}
```

Styling in react

- React's core ideas is to make an app's visual pieces self-contained and reusable.
- React encourages you to specify how your elements look right alongside the HTML and the JavaScript

```css
.letter {
 padding: 10px;
 margin: 10px;
 background-color: #FFDE00;
 color: #333;
 display: inline-block;
 font-family: monospace;
 font-size: 32px;
 text-align: center;
}
```

```jsx
// Remember class is a reserved word in JS, instead className
class Letter extends React.Component {
    render(){
        return (<div className="letter">{this.props.children}</div>);
    }
}
```

Styling in react

- React favors an inline approach for styling content that doesn't use CSS.

```
class Letter extends React.Component {
    render(){
        var letterStyle = {
          padding: 10,
          margin: 10,
          backgroundColor: "#FFDE00",
          color: "#333",
          display: "inline-block",
          fontFamily: "monospace",
          fontSize: 32,
          textAlign: "center"
        };

        return (<div style={letterStyle}>
                    {this.props.children}
                </div>
                );
    }
};
```

Composability

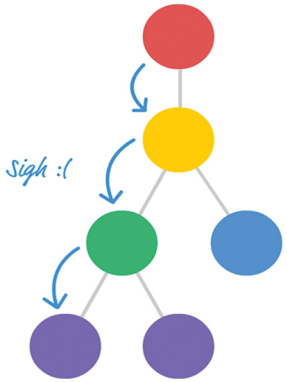
- You can combine components to create more complex components.

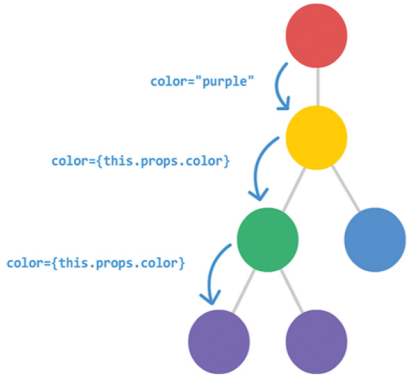
```
class Card extends React.Component {
    render() {
        var cardStyle = {
            height: 200,
            width: 150,
            padding: 0,
            backgroundColor: "#FFF",
            boxShadow: "0px 0px 5px #666"

        };
        return (<div style={cardStyle}>
                    <Square/>
                    <Label/>
                </div>);
    }
}
```

Transferring properties

- *React enforces a chain of command in which properties have to flow down from a parent component to an immediate child component.*
- This means you can't skip a layer of children when sending a property
- Your children can't send a property back up to a parent.
- All communication is one-way from the parent to the child.





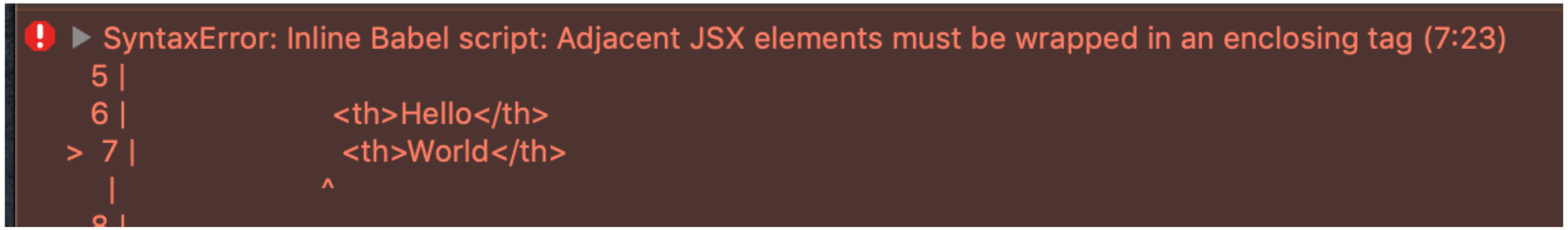
Transferring properties

- It’s better to use the spread operator for transferring properties
- Using the spread operator to transfer properties is convenient, and it’s a marked improvement over explicitly defining each property at each component.
- As created by the ES6/ES2015 committee, the spread operator is designed to work only on arrays and array like components (collections)

```
return (<div style={cardStyle}>
          <Square {...this.props}/>
          <Label  {...this.props}/>
        </div>);
```

JSX Considerations

- *JSX is not HTML*. It looks like HTML and behaves like it in many common scenarios, but it is ultimately designed to be translated into JavaScript.




```
class Columns extends React.Component {
    render(){
        return(

            <th>Hello</th>
            <th>World</th>

        );
    }
}
```

React.Fragment

- A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```

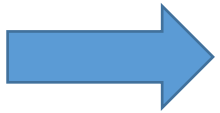
React comments

```
<!--
   Comments in JSX
   - As a child of a tag: {/* I am a child comment*/}
   - Inside a tag: /* */

   Recommendations:
    - HTML tags in lowecase
    - Component name capitalized
-->
```

State in React

- Adding local state to a Class
  - Add a class constructor that assigns the initial this.state:



```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

State in React

- Do not modify state directly

```
// Wrong
this.state.comment = 'Hello';
```

- Instead use setState()

```
// Correct
this.setState({comment: 'Hello'});
```

- The only place where you can assign this.state is the constructor.

State in React

- **State Updates May Be Asynchronous**
  - React may batch multiple setState() calls into a single update for performance.
  - Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

State in React

- **State Updates are Merged**
  - When you call setState(), React merges the object you provide into the current state.

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

- The merging is shallow,
  so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

State in React

- Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn’t care whether it is defined as a function or a class
- This is why state is often called local or encapsulated**. It is not accessible to any component other than the one that owns and sets it**.
- A component may choose to pass its state down as props to its child components:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

```
<FormattedDate date={this.state.date} />
```

Data to UI in React

- When you are building your apps, thinking in terms of props, state, components, JSX tags, render methods
- You are dealing with data in the form of JSON objects, arrays, and other data structures and you need to integrate data with UI

```
function showCircle(){
    var colors=["#393E41", "#E94F37", "#1C89BF", "#A1D363"];
    // Math.floor - Round a number downward to its nearest integer
    var ran = Math.floor(Math.random() * colors.length);

    return <Circle bgColor={colors[ran]}/>
}
```

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

Events in React

- Your **event handlers will be passed instances of SyntheticEvent**, a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers.
- Attributes of a **SyntheticEvent:**

```
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
```

Events in React

- Event pooling
  - The SyntheticEvent is pooled. This means that the SyntheticEvent object will be reused and all properties will be nullified after the event callback has been invoked.
  - This is for performance reasons. As such, you cannot access the event in an asynchronous way.
  - React normalizes events so that they have consistent properties across different browsers.
  - For a list of supported events
    - SEE: https://reactjs.org/docs/events.html

Handling events in React

- Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences:
  - React events are named using camelCase, rather than lowercase.
  - With JSX you pass a function as the event handler, rather than a string.

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Handling events in React

- Another difference is that you cannot return false to prevent default behavior in React. **You must call preventDefault explicitly**.

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Handling events in React (A)

- When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Handling events in React (B)

- If you are using the experimental public class fields syntax, you can use class fields to correctly bind callbacks:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Handling events in React (C)

- If you aren't using class fields syntax, you can use an arrow function in the callback: (Not recommended)

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Click me
      </button>
    );
  }
}
```

Passing arguments to event handlers

- The above two lines are equivalent, and use arrow functions and Function.prototype.bind respectively.
  - In both cases, the e argument representing the React event will be passed as a second argument after the ID.
  - With an arrow function, we have to pass it explicitly, but with bind any further arguments are automatically forwarded.

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

React Lifecyle

- Mounting
  - **Constructor** - The first thing that gets called is your component constructor, *if* your component is a class component. This does not apply to functional components.
  - **getDerivedStateFromProps -** When mounting, getDerivedStateFromProps is the last method called before rendering. You can use it to set state according to the initial props
  - **render -** Rendering does all the work. It returns the JSX of your actual component. When working with React, you'll spend most of your time here.
  - **componentDidMount -** After we've rendered our component for the first time, this method is called. If you need to load data, here's where you do it. Don't try to load data in the constructor or render or anything crazy

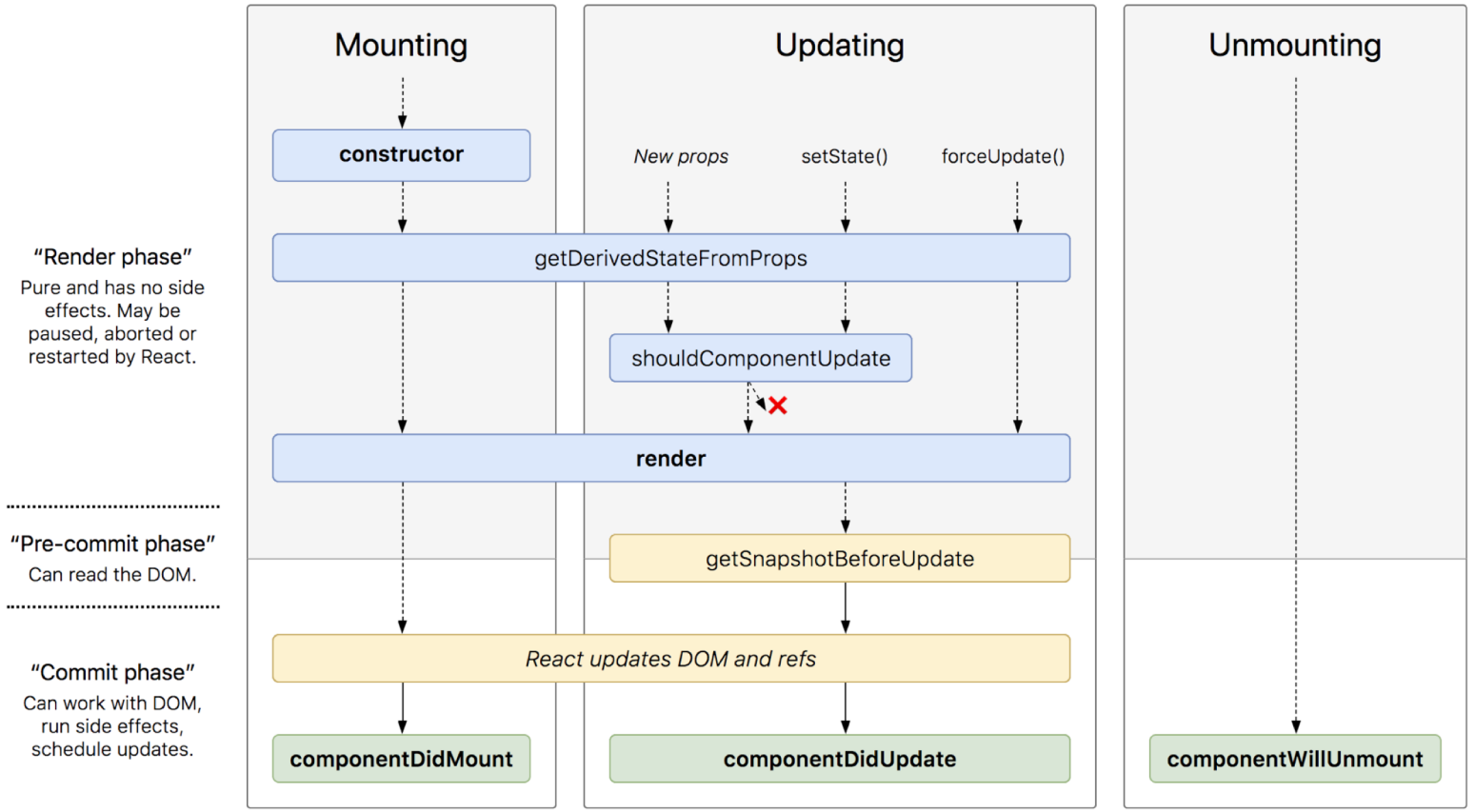
React Lifecyle

- Updating
  - **getDerivedStateFromProps -** If you need to update your state based on a prop changing, you can do it here by returning a new state object
  - **shouldComponentUpdate - S**hould always return a boolean — an answer to the question, “should I re-render?” Yes, little component, you should. The default is that it always returns true.
  - **render**
  - **getSnapshotBeforeUpdate -** Note it’s called between render and the updated component actually being propagated to the DOM. It exists as a last-chance-look at your component with its previous props and state. **You should either return null or a value from getSnapshotBeforeUpdate.**
  - **componentDidUpdate -** Now, our changes have been committed to the DOM. we have access to three things: the previous props, the previous state, and whatever value we returned from getSnapshotBeforeUpdate.

React Lifecyle

- Unmounting
  - **componentWillUnmount -** Your component is going to go away. Maybe forever. It’s very sad
- Errors
  - **getDerivedStateFromError -** Something broke. Not in your component itself, but one of its descendants
  - **componentDidCatch -** Very similar to the above, in that it is triggered when an error occurs in a child component. The difference is rather than updating state in response to an error, we can now perform any side effects, like logging the error

React Lifecyle



React Lifecyle

- In React 16.3 few lifecycle methods have been deprecated. For now, these methods are prefixed with UNSAFE_ and will be fully removed in the next major release.
- The methods that are deprecated are:
  - **componentWillMount**
    All the legacy use cases are covered in the constructor. This is renamed as UNSAFE_componentWillMount.
  - **componentWillReceiveProps**
    The new static method getDerivedStateFromProps is safe rewrite for this method and covers all the use cases of componentWillReceiveProps. The new name for this method is UNSAFE_componentWillReceiveProps.
  - **componentWillUpdate**
    The new method getSnapshotBeforeUpdate is safe rewrite for this method and covers all the use cases of componentWillUpdate.
    The new name for this method is UNSAFE_componentWillUpdate.

React and DOM

- In the typical React dataflow, props are the only way that parent components interact with their children.
- To modify a child, you re-render it with new props.
- However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow.
- The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

React and DOM

- Refs
  - Managing focus, text selection, or media playback.
  - Triggering imperative animations.
  - Integrating with third-party DOM libraries.
- Avoid using refs for anything that can be done declaratively.
- Refs are created using React.createRef() and attached to React elements via the ref attribute

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

React and DOM

- Accessing Refs
  - When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

```
const node = this.myRef.current;
```

  - The value of ref differs depending on the type of node:
    - When the ref attribute is used on an **HTML element**, receives the underlying DOM element as its current property.
    - When the ref attribute is used on a **custom class component**, the ref object receives the mounted instance of the component as its current
    - **You may not use the ref attribute on function components** because they don’t have instances

React and DOM

Adding Ref to a DOM element

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the constructor
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />

        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React and DOM

Adding Ref
to a class
component

If we wanted to wrap
the CustomTextInput above to
simulate it being clicked
immediately after mounting, we
could use a ref to get access to the
custom input and call
its focusTextInput method manually

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

```
class CustomTextInput extends React.Component {
  // ...
}
```

React and DOM

- React also supports another way to set refs called “callback refs”, which gives more fine-grain control over when refs are set and unset.
- Instead of passing a ref attribute created by createRef(), you pass a function. The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.

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

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // Focus the text input using the raw DOM API
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // autofocus the input on mount
    this.focusTextInput();
  }

  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React portals

- Portals provide a first-class way to **render children** into a DOM node that **exists outside the DOM hierarchy of the parent component**

```
ReactDOM.createPortal(child, container)
```

- The first argument (child) is any renderable React child, such as an element, string, or fragment. The second argument (container) is a DOM element.

React portals

- Normally, when you return an element from a component's render method, it's mounted into the DOM as a child of the nearest parent node

```
render() {
  // React mounts a new div and renders the children into it
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

React portals

- However, sometimes it’s useful to insert a child into a different location in the DOM:

```
render() {
  // React does *not* create a new div. It renders the children into
`domNode`.
  // `domNode` is any valid DOM node, regardless of its location in the DOM.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

Additional information

- React components - https://github.com/brillout/awesome-react-components
- React components that implement Google's material design - https://material-ui.com
- ReactDOM - https://reactjs.org/docs/react-dom.html
- Add React to a web site - https://reactjs.org/docs/add-react-to-a-website.html#add-react-in-one-minute
- Create a new react App - https://reactjs.org/docs/create-a-new-react-app.html
- UI Components - https://reactjs.org/community/ui-components.html

Additional information

- JSX Introduction - https://reactjs.org/docs/introducing-jsx.html
- Redux - https://redux.js.org
- Usage with react router - https://redux.js.org/advanced/usage-with-react-router
- Reactive extensions for react - https://github.com/Reactive-Extensions/RxJS
- Lifecycle methods - https://blog.bitsrc.io/react-16-lifecycle-methods-how-and-when-to-use-them-f4ad31fb2282
- Understand React v16.4+ New Component Lifecycle methods - https://blog.bitsrc.io/understanding-react-v16-4-new-component-lifecycle-methods-fa7b224efd7d