[Introducing JSX](https://reactjs.org/docs/introducing-jsx.html)

# JSX produces React “elements”

# React [separates concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) with loosely coupled units called “components”

# React [doesn’t require](https://reactjs.org/docs/react-without-jsx.html) using JSX,

# You can put any valid [JavaScript expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) inside the curly braces in JSX.

**e.g const name = 'Josh Perez';**

**const element = <h1>Hello, {name}</h1>;**

# 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

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

# JSX Prevents Injection Attacks. It is safe to embed user input in JSX:

***const title = response.potentiallyMaliciousInput;***

***// This is safe:***

***const element = <h1>{title}</h1>;***

By default, React DOM [escapes](https://stackoverflow.com/questions/7381974/which-characters-need-to-be-escaped-on-html) 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)](https://en.wikipedia.org/wiki/Cross-site_scripting) attacks.

# Babel compiles JSX down to React.createElement() calls.

**const element = (**

**<h1 className="greeting">**

**Hello, world!**

**</h1>**

**);**

**OR**

**const element = React.createElement(**

**'h1',**

**{className: 'greeting'},**

**'Hello, world!'**

**);**

React.createElement() essentially it creates an object like this:

**// Note: this structure is simplified**

**const element = {**

**type: 'h1',**

**props: {**

**className: 'greeting',**

**children: 'Hello, world!'**

**}**

**};**

These objects are called “React elements”. React reads these objects and uses them to construct the DOM and keep it up to date.

[Rendering Elements](https://reactjs.org/docs/rendering-elements.html)

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

# React elements are [immutable](https://en.wikipedia.org/wiki/Immutable_object). Once you create an element, you can’t change its children or attributes.

# With our knowledge so far, the only way to update the UI is to create a new element, and pass it to [ReactDOM.render()](https://reactjs.org/docs/react-dom.html#render).

**function tick() {**

**const element = (**

**<div>**

**<h1>Hello, world!</h1>**

**<h2>It is {new Date().toLocaleTimeString()}.</h2>**

**</div>**

**);**

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

**}**

**setInterval(tick, 1000);**

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents have changed gets updated by React DOM.

# React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

[Components and Props](https://reactjs.org/docs/components-and-props.html)

# Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

# Components return React elements describing what should appear on the screen.

# When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

# Whether you declare a component [as a function or a class](https://reactjs.org/docs/components-and-props.html#function-and-class-components), it must never modify its own props.

# function sum(a, b) {

return a + b;

}

Such functions are called [“pure”](https://en.wikipedia.org/wiki/Pure_function) because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:

function withdraw(account, amount) {

account.total -= amount;

}

# **All React components must act like pure functions with respect to their props.**

[State and Lifecycle](https://reactjs.org/docs/state-and-lifecycle.html)

# You can convert a function component to a class in five steps:

1. Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), with the same name, that extends React.Component.
2. Add a single empty method to it called render().
3. Move the body of the function into the render() method.
4. Replace props with this.props in the render() body.
5. Delete the remaining empty function declaration.

# Class components should always call the base constructor with props.

**constructor(props) {**

**super(props); this.state = {date: new Date()};**

**}**

# In applications with many components, it’s very important to free up resources taken by the components when they are destroyed.

# The componentDidMount() method runs after the component output has been rendered to the DOM.

**componentDidMount() {**

**this.timerID = setInterval(**

**() => this.tick(),**

**1000**

**);**

**}**

**componentWillUnmount() {**

**clearInterval(this.timerID);**

**}**

# If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

# **Do Not Modify State Directly**

will not re-render a component

# **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.

# second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

**this.setState((state, props) => ({**

**counter: state.counter + props.increment**

**}));**

# When you call setState(), React merges the object you provide into the current state.

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

# 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.

[Handling Events](https://reactjs.org/docs/handling-events.html)

# React events are named using camelCase, rather than lowercase.

# With JSX you pass a function as the event handler, rather than a string.

# For example, the HTML:

**<button onclick="activateLasers()">**

**Activate Lasers**

**</button>**

is slightly different in React:

**<button onClick={activateLasers}> Activate Lasers**

**</button>**

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

# React events do not work exactly the same as native events. See the [SyntheticEvent](https://reactjs.org/docs/events.html) reference guide to learn more.

# When using React, you generally don’t need to call addEventListener to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

# You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not [bound](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind) by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.

// This syntax ensures `this` is bound within handleClick. // Warning: this is \*experimental\* syntax.

**handleClick = () => { console.log('this is:', this); }**

# **render() {**

**// This syntax ensures `this` is bound within handleClick return ( <button onClick={() => this.handleClick()}>**

**Click me**

**</button>**

**);**

**}**

**The problem with this syntax is that a different callback is created each time this component renders.**

**In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering.**

**We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.**

# **Passing Arguments to Event Handlers**

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

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

The above two lines are equivalent, and use [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) and [Function.prototype.bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/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.

[Conditional Rendering](https://reactjs.org/docs/conditional-rendering.html)

# In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

**render() {**

**const isLoggedIn = this.state.isLoggedIn;**

**let button;**

**if (isLoggedIn) { button = <LogoutButton onClick={this.handleLogoutClick} />; }**

**else { button = <LoginButton onClick={this.handleLoginClick} />; }**

**return (**

**<div>**

**<Greeting isLoggedIn={isLoggedIn} /> {button}**

**</div>**

**);**

**}**

# You may [embed expressions in JSX](https://reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx) by wrapping them in curly braces. This includes the JavaScript logical && operator.

**{ unreadMessages.length > 0 && <h2> You have {unreadMessages.length} unread messages. </h2>**

**}**

It works because in JavaScript, true && expression always evaluates to expression, and false && expression always evaluates to false.

# Returning null from a component’s render method ( basically don’t render that component If you return null from render method ) does not affect the firing of the component’s lifecycle methods. For instance componentDidUpdate will still be called.

#

#

#

#

#

#

#

#

#

#

#

#