**REACT**

One important thing to know about nested JSX is that it must return a single element.

This one parent element would wrap all of the other levels of nested elements.

For instance, several JSX elements written as siblings with no parent wrapper element will not transpile.

Here's an example:

**Valid JSX:**

<div>  
  <p>Paragraph One</p>  
  <p>Paragraph Two</p>  
  <p>Paragraph Three</p>  
</div>

**Invalid JSX:**

<p>Paragraph One</p>  
<p>Paragraph Two</p>  
<p>Paragraph Three</p>

When rendering multiple elements like this, you can wrap them all in parentheses, but it's not strictly required. Also notice this challenge uses a divtag to wrap all the child elements within a single parent element. If you remove the div, the JSX will no longer transpile. Keep this in mind, since it will also apply when you return JSX elements in React components.

RENDERING HTML ELEMENTS TO THE DOM

So far, you've learned that JSX is a convenient tool to write readable HTML within JavaScript. With React, we can render this JSX directly to the HTML DOM using React's rendering API known as ReactDOM.

ReactDOM offers a simple method to render React elements to the DOM which looks like this: ReactDOM.render(componentToRender, targetNode), where the first argument is the React element or component that you want to render, and the second argument is the DOM node that you want to render the component to.

As you would expect, ReactDOM.render()must be called after the JSX element declarations, just like how you must declare variables before using them.

JSX SYNTAX

CLASS

One key difference in JSX is that you can no longer use the word class to define HTML classes. This is because class is a reserved word in JavaScript. Instead, JSX uses className. In fact, the naming convention for all HTML attributes and event references in JSX become camelCase. For example, a click event in JSX is onClick, instead of onclick.

SELF CLOSING TAGS

In JSX, the rules are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed. The line-break tag, for example, must always be written as <br />in order to be valid JSX that can be transpiled. A <div>, on the other hand, can be written as <div />or <div></div>. The difference is that in the first syntax version there is no way to include anything in the <div />. You will see in later challenges that this syntax is useful when rendering React components.

CREATING STATELESS COMPONENTS

Components are the core of React. Everything in React is a component and here you will learn how to create one.

There are two ways to create a React component. The first way is to use a JavaScript function. Defining a component in this way creates a *stateless functional component*. The concept of state in an application will be covered in later challenges. For now, think of a stateless component as one that can receive data and render it, but does not manage or track changes to that data.

To create a component with a function, you simply write a JavaScript function that returns either JSX or null. One important thing to note is that React requires your function name to begin with a capital letter. Here's an example of a stateless functional component that assigns an HTML class in JSX:

// After being transpiled, the <div> will have a CSS class of 'customClass'  
const DemoComponent = function() {  
  return (  
    <div className='customClass' />  
  );  
};

Because a JSX component represents HTML, you could put several components together to create a more complex HTML page. This is one of the key advantages of the component architecture React provides. It allows you to compose your UI from many separate, isolated components. This makes it easier to build and maintain complex user interfaces.

The second way to define a React component is with the ES6 class syntax. In the following example, Kitten extends React.Component:

class Kitten extends React.Component {  
  constructor(props) {  
    super(props);  
  }  
  
  render() {  
    return (  
      <h1>Hi</h1>  
    );  
  }  
}

REACT COMPONENTS TO THE DOM

React components are passed into ReactDOM.render() a little differently than JSX elements. For JSX elements, you pass in the name of the element that you want to render. However, for React components, you need to use the same syntax as if you were rendering a nested component, for example ReactDOM.render(<ComponentToRender />, targetNode). You use this syntax for both ES6 class components and functional components.

PROPS

In React, you can pass props, or properties, to child components. Say you have an App component which renders a child component called Welcome that is a stateless functional component. You can pass Welcome a user property by writing:

<App>  
  <Welcome user='Mark' />  
</App>

You use **custom HTML attributes** that React provides support for to pass the property user to the component Welcome. Since Welcome is a stateless functional component, it has access to this value like so:

const Welcome = (props) => <h1>Hello, {props.user}!</h1>

It is standard to call this value props and when dealing with stateless functional components, you basically consider it as an argument to a function which returns JSX. You can access the value of the argument in the function body. With class components, you will see this is a little different.

USE PROPTYPES TO DEFINER THE EXPECTED PROPS

React provides useful type-checking features to verify that components receive props of the correct type. For example, your application makes an API call to retrieve data that you expect to be in an array, which is then passed to a component as a prop. You can set propTypeson your component to require the data to be of type array. This will throw a useful warning when the data is of any other type.

It's considered a best practice to set propTypes when you know the type of a prop ahead of time. You can define a propTypes property for a component in the same way you defined defaultProps. Doing this will check that props of a given key are present with a given type. Here's an example to require the type function for a prop called handleClick:

MyComponent.propTypes = { handleClick: PropTypes.func.isRequired }

ACCESS PROPS USING THIS

The last several challenges covered the basic ways to pass props to child components. But what if the child component that you're passing a prop to is an ES6 class component, rather than a stateless functional component? The ES6 class component uses a slightly different convention to access props.

Anytime you refer to a class component within itself, you use the this keyword. To access props within a class component, you preface the code that you use to access it with this. For example, if an ES6 class component has a prop called data, you write {this.props.data}in JSX.

REVIEW OF STATEFULNESS AND COMPONENTS

A stateless functional component is any function you write which accepts props and returns JSX. A stateless component, on the other hand, is a class that extends React.Component, but does not use internal state (covered in the next challenge). Finally, a stateful component is any component that does maintain its own internal state. You may see stateful components referred to simply as components or React components.

A common pattern is to try to minimize statefulness and to create stateless functional components wherever possible. This helps contain your state management to a specific area of your application. In turn, this improves development and maintenance of your app by making it easier to follow how changes to state affect its behavior.

STATE IN REACT

One of the most important topics in React is state. State consists of any data your application needs to know about, that can change over time. You want your apps to respond to state changes and present an updated UI when necessary. React offers a nice solution for the state management of modern web applications.

You create state in a React component by declaring a state property on the component class in its constructor. This initializes the component with state when it is created. The state property must be set to a JavaScript object. Declaring it looks like this:

this.state = {  
  // describe your state here  
}

You have access to the state object throughout the life of your component. You can update it, render it in your UI, and pass it as props to child components. The state object can be as complex or as simple as you need it to be. Note that you must create a class component by extending React.Component in order to create state like this.

STATE RENDERING IN THE UI

Once you define a component's initial state, you can display any part of it in the UI that is rendered. If a component is stateful, it will always have access to the data in statein its render()method. You can access the data with this.state.

If you want to access a state value within the returnof the render method, you have to enclose the value in curly braces.

Stateis one of the most powerful features of components in React. It allows you to track important data in your app and render a UI in response to changes in this data. If your data changes, your UI will change. React uses what is called a virtual DOM, to keep track of changes behind the scenes. When state data updates, it triggers a re-render of the components using that data - including child components that received the data as a prop. React updates the actual DOM, but only where necessary. This means you don't have to worry about changing the DOM. You simply declare what the UI should look like.

Note that if you make a component stateful, no other components are aware of its state. Its stateis completely encapsulated, or local to that component, unless you pass state data to a child component as props. This notion of encapsulated state is very important because it allows you to write certain logic, then have that logic contained and isolated in one place in your code.

RENDERING STATE ANOTHER WAY

There is another way to access state in a component. In the render()method, before the return statement, you can write JavaScript directly. For example, you could declare functions, access data from state or props, perform computations on this data, and so on. Then, you can assign any data to variables, which you have access to in the return statement.

Set State with this.setState

The previous challenges covered component state and how to initialize state in the constructor. There is also a way to change the component's state. React provides a method for updating component state called setState. You call the setState method within your component class like so: this.setState(), passing in an object with key-value pairs. The keys are your state properties and the values are the updated state data. For instance, if we were storing a username in state and wanted to update it, it would look like this:

this.setState({  
  username: 'Lewis'  
});

React expects you to never modify state directly, instead always use this.setState()when state changes occur.

BIND THIS TO A CLASS METHOD

In addition to setting and updating state, you can also define methods for your component class. A class method typically needs to use the this keyword so it can access properties on the class (such as state and props) inside the scope of the method. There are a few ways to allow your class methods to access this.

One common way is to explicitly bind this in the constructor so this becomes bound to the class methods when the component is initialized. You may have noticed the last challenge used this.handleClick = this.handleClick.bind(this)for its handleClick method in the constructor. Then, when you call a function like this.setState()within your class method, this refers to the class and will not be undefined.

USE STATE TO TOGGLE AN ELEMENT

You can use state in React applications in more complex ways than what you've seen so far. One example is to monitor the status of a value, then render the UI conditionally based on this value. There are several different ways to accomplish this, and the code editor shows one method.

Pass State as Props to Child Components

You saw a lot of examples that passed props to child JSX elements and child React components in previous challenges. You may be wondering where those props come from. A common pattern is to have a stateful component containing the stateimportant to your app, that then renders child components. You want these components to have access to some pieces of that state, which are passed in as props.

For example, maybe you have an App component that renders a Navbar, among other components. In your App, you have state that contains a lot of user information, but the Navbar only needs access to the user's username so it can display it. You pass that piece of stateto the Navbar component as a prop.

This pattern illustrates some important paradigms in React. The first is *unidirectional data flow*. State flows in one direction down the tree of your application's components, from the stateful parent component to child components. The child components only receive the state data they need.

The second is that complex stateful apps can be broken down into just a few, or maybe a single, stateful component. The rest of your components simply receive state from the parent as props, and render a UI from that state.

It begins to create a separation where state management is handled in one part of code and UI rendering in another. This principle of separating state logic from UI logic is one of React's key principles. When it's used correctly, it makes the design of complex, stateful applications much easier to manage.

PASSING CALLBACK AS PROPS

You can pass state as props to child components, but you're not limited to passing data. You can also pass handler functions or any method that's defined on a React component to a child component. This is how you allow child components to interact with their parent components. You pass methods to a child just like a regular prop. It's assigned a name and you have access to that method name under this.props in the child component.

LIFECYCLE METHODS

React components have several special methods that provide opportunities to perform actions at specific points in the lifecycle of a component. These are called lifecycle methods, or lifecycle hooks, and allow you to catch components at certain points in time. This can be before they are rendered, before they update, before they receive props, before they unmount, and so on. Here is a list of some of the main lifecycle methods:

componentWillMount()

componentDidMount()

Most web developers, at some point, need to call an API endpoint to retrieve data. If you're working with React, it's important to know where to perform this action.

The best practice with React is to place API calls or any calls to your server in the lifecycle method componentDidMount(). This method is called after a component is mounted to the DOM. Any calls to setState()here will trigger a re-rendering of your component. When you call an API in this method, and set your state with the data that the API returns, it will automatically trigger an update once you receive the data.

ADDING EVENT LISTENERS

The componentDidMount() method is also the best place to attach any event listeners you need to add for specific functionality. React provides a synthetic event system which wraps the native event system present in browsers. This means that the synthetic event system behaves exactly the same regardless of the user's browser - even if the native events may behave differently between different browsers.

You've already been using some of these synthetic event handlers such as onClick(). React's synthetic event system is great to use for most interactions you'll manage on DOM elements. However, if you want to attach an event handler to the document or window objects, you have to do this directly.

componentWillReceiveProps()

componentDidUpdate()

## React: Manage Updates with Lifecycle Methods

Another lifecycle method is componentWillReceiveProps() which is called whenever a component is receiving new props. This method receives the new props as an argument, which is usually written as nextProps. You can use this argument and compare with this.propsand perform actions before the component updates. For example, you may call setState()locally before the update is processed.

Another method is componentDidUpdate(), and is called immediately after a component re-renders. Note that rendering and mounting are considered different things in the component lifecycle. When a page first loads, all components are mounted and this is where methods like componentWillMount()and componentDidMount()are called. After this, as state changes, components re-render themselves. The next challenge covers this in more detail.

## Manage Updates with Lifecycle Methods

This challenge has you creating a couple lifecycle functions, componentWillUpdate and ComponentWillReceiveProps. You will be provided with another function called componentDidUpdate. We’ll discuss how you use them at each stage of the component lifecycle and why you should use them when you are checking different stages of your component.

Lets talk about the functions and how you will be using them. Component lifecycles can be broken down into 4 stages. Initlization -> Mounting -> Updating -> Unmounting. The components that you will work with are going to fall within the Updating stage.

The progression in which these functions are called are as follows: componentWillReceiveProps -> componentWillUpdate -> componentDidUpdate

When you create componentWillReceiveProps, this function will check to see if there are new props being received. If the component did receive new props then the function will be called and within the block you can compare the two prop states. The function will take in an argument typically named nextProps and will compare it to this.props. The challenge has you creating this function using the passed argument nextProps. See provided function below.

Next in the component lifecycle componentWillUpdate will be called, this function will check to see if there has been any updates to props or state and will be called before the component renders. The challenge has already provided you with this function and it logs out “Component is about to update.”

Once the component passes through the componentWillUpdate phase and the component renders, componentDidUpdate will be called. At this stage you can call this.setState to update any state chanegs that occurred during the first two phases. Note: you can only call setState if you wrap within a condition. Since this challenge only has you scratching the surface they would like you to log out that the “Component has updated.”

Once you have implemented all the lifecycle functions you should see some console logs being displayed. First, you will see componentWillReceiveProps send you this.props and nextProps. Next, you will see a console log letting you know that componentWillUpdate. Lastly, after the component renders it will call the componentDidUpdate and will log out “Component has updated.”

Note: The components that you are creating have been deprecated and will be available to use until version 17. You can find more information about these functions in the resource section below.

shouldComponentUpdate()

So far, if any component receives new stateor new props, it re-renders itself and all its children. This is usually okay. But React provides a lifecycle method you can call when child components receive new stateor props, and declare specifically if the components should update or not. The method is shouldComponentUpdate(), and it takes nextPropsand nextStateas parameters.

This method is a useful way to optimize performance. For example, the default behavior is that your component re-renders when it receives new props, even if the propshaven't changed. You can use shouldComponentUpdate()to prevent this by comparing the props. The method must return a booleanvalue that tells React whether or not to update the component. You can compare the current props (this.props) to the next props (nextProps) to determine if you need to update or not, and return trueor falseaccordingly.

componentWillUpdate()

componentWillUnmount()

INLINE STYLING(React17)

If you import styles from a stylesheet, it isn't much different at all. You apply a class to your JSX element using the classNameattribute, and apply styles to the class in your stylesheet. Another option is to apply ***inline*** styles, which are very common in ReactJS development.

You apply inline styles to JSX elements similar to how you do it in HTML, but with a few JSX differences. Here's an example of an inline style in HTML:

<div style="color: yellow; font-size: 16px">Mellow Yellow</div>

JSX elements use the styleattribute, but because of the way JSX is transpiled, you can't set the value to a string. Instead, you set it equal to a JavaScript object. Here's an example:

<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>

Notice how we camelCase the "fontSize" property? This is because React will not accept kebab-case keys in the style object. React will apply the correct property name for us in the HTML. As a rule, any hyphenated style properties are written using camel case in JSX.

RENDER WITH IF/ELSE CONDITIONS

Another application of using JavaScript to control your rendered view is to tie the elements that are rendered to a condition. When the condition is true, one view renders. When it's false, it's a different view. You can do this with a standard if/else statement in the render()method of a React component.

CONCISE CONDITIONALS

If you write a lot of else if statements to return slightly different UIs, you may repeat code which leaves room for error. Instead, you can use the &&logical operator to perform conditional logic in a more concise way. This is possible because you want to check if a condition is true, and if it is, return some markup. Here's an example:

{condition && <p>markup</p>}

If the condition is true, the markup will be returned. If the condition is false, the operation will immediately return false after evaluating the condition and return nothing. You can include these statements directly in your JSX and string multiple conditions together by writing &&after each one. This allows you to handle more complex conditional logic in your render()method without repeating a lot of code.

TERNARY OPERATOR

The ternary operator is often utilized as a shortcut for if/else statements in JavaScript. They're not quite as robust as traditional if/else statements, but they are very popular among React developers. One reason for this is because of how JSX is compiled, if/else statements can't be inserted directly into JSX code.

Ternary expressions can be an excellent alternative if you want to implement conditional logic within your JSX. Recall that a ternary operator has three parts, but you can combine several ternary expressions together. Here's the basic syntax:

condition ? expressionIfTrue : expressionIfFalse

**RENDER CONDITIONALLY FROM PROPS**

So far, you've seen how to use if/else, &&,nulland the ternary operator (condition ? expressionIfTrue : expressionIfFalse) to make conditional decisions about what to render and when. However, there's one important topic left to discuss that lets you combine any or all of these concepts with another powerful React feature: props. Using props to conditionally render code is very common with React developers — that is, they use the value of a given prop to automatically make decisions about what to render.

In this challenge, you'll set up a child component to make rendering decisions based on props. You'll also use the ternary operator, but you can see how several of the other concepts that were covered in the last few challenges might be just as useful in this context.

 Change Inline CSS Conditionally Based on Component State(React 22.js)

At this point, you've seen several applications of conditional rendering and the use of inline styles. Here's one more example that combines both of these topics. You can also render CSS conditionally based on the state of a React component. To do this, you check for a condition, and if that condition is met, you modify the styles object that's assigned to the JSX elements in the render method.

This paradigm is important to understand because it is a dramatic shift from the more traditional approach of applying styles by modifying DOM elements directly (which is very common with jQuery, for example). In that approach, you must keep track of when elements change and also handle the actual manipulation directly. It can become difficult to keep track of changes, potentially making your UI unpredictable. When you set a style object based on a condition, you describe how the UI should look as a function of the application's state. There is a clear flow of information that only moves in one direction. This is the preferred method when writing applications with React.

USING ARRAY.MAP(React22)

Conditional rendering is useful, but you may need your components to render an unknown number of elements. Often in reactive programming, a programmer has no way to know what the state of an application is until runtime, because so much depends on a user's interaction with that program. Programmers need to write their code to correctly handle that unknown state ahead of time. Using Array.map()in React illustrates this concept.

For example, you create a simple "To Do List" app. As the programmer, you have no way of knowing how many items a user might have on their list. You need to set up your component to ***dynamically render*** the correct number of list elements long before someone using the program decides that today is laundry day.

UNIQUE KEY ATTRIBUTE(React23)

The last challenge showed how the map method is used to dynamically render a number of elements based on user input. However, there was an important piece missing from that example. When you create an array of elements, each one needs a keyattribute set to a unique value. React uses these keys to keep track of which items are added, changed, or removed. This helps make the re-rendering process more efficient when the list is modified in any way. Note that keys only need to be unique between sibling elements, they don't need to be globally unique in your application.

REACT ON THE SERVER

So far, you have been rendering React components on the client. Normally, this is what you will always do. However, there are some use cases where it makes sense to render a React component on the server. Since React is a JavaScript view library and you can run JavaScript on the server with Node, this is possible. In fact, React provides a renderToString()method you can use for this purpose.

There are two key reasons why rendering on the server may be used in a real world app. First, without doing this, your React apps would consist of a relatively empty HTML file and a large bundle of JavaScript when it's initially loaded to the browser. This may not be ideal for search engines that are trying to index the content of your pages so people can find you. If you render the initial HTML markup on the server and send this to the client, the initial page load contains all of the page's markup which can be crawled by search engines.

Second, this creates a faster initial page load experience because the rendered HTML is smaller than the JavaScript code of the entire app. React will still be able to recognize your app and manage it after the initial load.

**REDUX**

* In Redux, there is a single state object that's responsible for the entire state of your application. This means if you had a React app with ten components, and each component had its own local state, the entire state of your app would be defined by a single state object housed in the Redux store. This is the first important principle to understand when learning Redux: the Redux store is the single source of truth when it comes to application state.
* This also means that any time any piece of your app wants to update state, it **must** do so through the Redux store. The unidirectional data flow makes it easier to track state management in your app.

Challenge 2

// The Redux store object provides several methods that allow you to interact with it. For example, you can retrieve the current state held in the Redux store object with the getState() method.

Challenge 3:

Since Redux is a state management framework, updating state is one of its core tasks. In Redux, all state updates are triggered by dispatching actions. An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly. Sometimes a Redux action also carries some data. For example, the action carries a username after a user logs in. While the data is optional, actions must carry a type property that specifies the 'type' of action that occurred.

Think of Redux actions as messengers that deliver information about events happening in your app to the Redux store. The store then conducts the business of updating state based on the action that occurred.

Challenge 4:

After creating an action, the next step is sending the action to the Redux store so it can update its state. In Redux, you define action creators to accomplish this. An action creator is simply a JavaScript function that returns an action. In other words, action creators create objects that represent action events.

## Redux: Dispatch an Action Event

Dispatch method is what you use to dispatch actions to the Redux store. Calling store.dispatch()and passing the value returned from an action creator sends an action back to the store.

Recall that action creators return an object with a type property that specifies the action that has occurred. Then the method dispatches an action object to the Redux store. Based on the previous challenge's example, the following lines are equivalent, and both dispatch the action of type LOGIN:

store.dispatch(actionCreator());  
store.dispatch({ type: 'LOGIN' });

Challenge 6:

After an action is created and dispatched, the Redux store needs to know how to respond to that action. This is the job of a reducer function. Reducers in Redux are responsible for the state modifications that take place in response to actions. A reducer takes state and action as arguments, and it always returns a new state. It is important to see that this is the **only** role of the reducer. It has no side effects — it never calls an API endpoint and it never has any hidden surprises. The reducer is simply a pure function that takes state and action, then returns new state.

Another key principle in Redux is that state is read-only. In other words, the reducer function must **always** return a new copy of state and never modify state directly. Redux does not enforce state immutability, however, you are responsible for enforcing it in the code of your reducer functions. You'll practice this in later challenges.

Challenge 7:

You can tell the Redux store how to handle multiple action types. Say you are managing user authentication in your Redux store. You want to have a state representation for when users are logged in and when they are logged out. You represent this with a single state object with the property authenticated. You also need action creators that create actions corresponding to user login and user logout, along with the action objects themselves.

Challenge 8:

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used. You can refactor the code you're working with to write the action types as const declarations..