**Build Apps Using React: Advanced Features in React**

In this 19-video course, you will discover advanced React features such as lists and keys, including lists without keys, unique keys for lists, and correct key usage, and other topics. Learners begin with a look at the error that occurs when list elements are rendered without keys, then study the unique keys for list elements. Learn about the right usage of keys in components; how to render components using conditional evaluation; how to perform conditional rendering using variables; and how to evaluate conditions by using inline logical operators. Explore how to perform rendering with ternary operators; the use of local state in components; and the disadvantages of storing local state. Learn about removing state from lower-level components, and lifting state up to higher-level components. Delve into code reuse by using inheritance and code reuse by using composition. Examine how to use composition to customize child elements and how to create special components using composition. Explore the use of global properties without context, and finally, the use of context to specify global properties.

**Course Overview**

[Video description begins] *Topic title: Course Overview.* [Video description ends]

Hi, and welcome to this course in the React learning path, advanced features in React. My name is Janani Ravi, and I will be your instructor for this course.

[Video description begins] *Your host for this session is Janani Ravi. She is a Software engineer and big data expert.* [Video description ends]

A little about myself first. I did my masters from Stanford University, and have worked at various companies including Google. I presently work for Loonycorn, a studio for high quality video content. React is a declarative component based JavaScript library, which originated at Facebook, and is now very widely used and supported by a vibrant open source community. React makes use of a clever architectural design known as the virtual DOM, which makes it ideal for apps that work with fast-changing data.

In addition, React uses a declarative extension to the JavaScript language known as JSX making it easy to control component rendering. This course covers features such as lists and keys. The correct way to specify key attributes for list elements. Conditional rendering of components using inline and ternary operators. Lifting state up the component tree. And code reuse using composition rather than inheritance. We will also discuss how you can modify components using context for global properties.

Once you're done with this course, you will be able to explain the error that occurs when list elements are rendered without keys. You'll see how you can specify unique keys for list elements and sketch the right usage of keys and components. You will also be able to evaluate conditions using inline logical operators. Perform a rendering using ternary operators. And reuse code using both composition as well as inheritance.

**Lists without Keys**

[Video description begins] *Topic title: Lists without Keys. Your host for this session is Janani Ravi.* [Video description ends]

In this demo and in the next few demos that follow, we'll revisit a concept that we've studied earlier. That of associating keys with sibling elements so that React knows which element needs to be updated.

[Video description begins] *A Sublime text editor opens. A Folders pane is present on the left. It has a folder named ListsAndKeys. It has a sub folder named Lists. It contains two files: Lists.html and Lists.js. In the editor pane, two tabs are present: Lists.html and Lists.js. Lists.html is currently open. It has some code lines.* [Video description ends]

We'll gradually introduce this concept once again. And we'll discuss in detail what you should be using as keys for your React components. Let's head over to list.js and let's set up some code to display a list.

[Video description begins] *She shifts to the tab titled Lists.js. It has some code lines.* [Video description ends]

I've defined here in this JS file a component called NamesList which extends the base React.Component class. Within the render function for this component, we simply display a list of items. The list of items that we want displayed is passed in as a part of the props to this component. this.props.names gives us a list of names that we'll display. I invoke the map function on this list. Remember the map function allows us to specify a transformation on each element of the original NamesList. The input argument to the map function is a function that defines that transformation.

[Video description begins] *She points at code line 4. It reads: var listItems = this.props.names.map(.* [Video description ends]

For each element in the original NamesList we'll convert the name to uppercase and enclose it within list item tags or LI tags.

[Video description begins] *She points at code line 5. It reads: (name) => <li>{name.toUpperCase()}</li>.* [Video description ends]

The resulting list items JavaScript list will be a list of LI tags or LI elements. And then finally we'll enclose these LI elements into an unordered list.

[Video description begins] *She points at code line 9. It reads: <ul className = "names">{listItems}</ul>.* [Video description ends]

This unordered list is what we've defined using a UL tag and that is what is returned from this render function. The names that I want rendered to screen are present in this names constant here Esther, Lily, Candace, Edward and Sidney.

[Video description begins] *She highlights code line 15.* [Video description ends]

I then invoke ReactDOM.render, pass in the names list component. And the props that I specify for names is this list of names.

[Video description begins] *She points at code line 17. It reads: ReactDOM.render(<NamesList names={names} />,.* [Video description ends]

Now those who have been paying attention may have noticed that I have a number of LI tags that are sibling elements, but I haven't associated a unique key with each tag. And you know that this is not really right and this will trigger a warning. Let's confirm this. Let's switch over to our Chrome browser and click through to list and list.html.

[Video description begins] *A page titled Index of/Project/React/ListsAndKeys/ opens in a browser. On the left, a section titled Index of/Project/React/ListsAndKeys/ is present. It has a link for List. On the right, three tabs are present: Elements, Console, and Sources. Console is currently selected.* [Video description ends]

Let's click on this HTML file and take a look at the code that we just wrote.

[Video description begins] *A link for Lists.html appears.* [Video description ends]

Our unordered list was rendered successfully, Esther, Lily, Candace, Edward and Sidney all in uppercase. But there is a bright red warning off to the right. The warning makes it very explicit that each child in an array or iterator should have a unique key property that we need to specify. The render method of NamesList does not specify this key property for the LI tags. This key property is not associated with the HTML. If you inspect this list and view the divs that are associated with each of these list items, the LI tag will look perfectly fine.

[Video description begins] *In the left, she clicks on Esther. A context menu appears. She clicks on Inspect. On the right, Elements tab get selected. It has some lines of code. She highlights 5 code lines. Line 1 is: <li>ESTHER</li>== $0. Line 2 is: <li>LILY</li>. Line 3 is:< li>CANDACE</li>. Line 4 is: <li>EDWARD</li>. Line 5 is:<li>SIDNEY</li>.* [Video description ends]

The warning is from React. React expects these unique keys to be present to uniquely identify these list items to make updates more performant, so that only the items that have been updated will be re-rendered.

**Unique Keys for Lists**

[Video description begins] *Topic title: Unique Keys for Lists. Your host for this session is Janani Ravi.* [Video description ends]

We left the last demo off at a point where we displayed a list using React, but we hadn't specified a unique key for each list element. And React definitely needs one in order to re-render elements that have changed in a performant manner.

[Video description begins] *A Sublime text editor opens. A Folders pane is present on the left. It has a folder named ListsWithKeys. It expands to show two files: ListsKey.html and ListsKey.js. In the editor pane, two tabs are present: ListsKey.html and ListsKey.js. Currently, the ListsKey.html tab is open. It has some code lines.* [Video description ends]

We'll fix that in this demo. The HTML for this is all of the same stuff that we are expecting. Let's head over to ListKey.js and take a look at the code.

[Video description begins] *She shifts to the tab titled ListsKey.js. It has some code lines.* [Video description ends]

The code here is only a slight modification of the previous demo. I have a constant, names, which is a JavaScript list of names. Esther, Lily, Candace, Edward, and Sidney.

[Video description begins] *She highlights code line 16.* [Video description ends]

I have a component called NamesList, which extends the React base Component class.

[Video description begins] *She highlights code line 1.* [Video description ends]

And within the render function, we access this.props.names and invoke the map function on the list and perform a transformation which generates a list item for each item in the names list.

[Video description begins] *She points at code line 5. It reads: var listItems = this.props.names.map(.* [Video description ends]

We also convert the name to uppercase. Now, I have specified a key here. Observe that for every list item, I have a key attribute. But I have set all of the keys equal to the same value, 1. Every li element in this list has a key, but the key is not unique.

[Video description begins] *She points at code line 6. It reads: (name) => <li key={1}>{name.toUpperCase()}</li>.* [Video description ends]

All of the keys for all of the elements of this list will be equal to 1. And this, clearly, is not what React expects, as we shall soon see. Go ahead, save this file, and let's head over to our Chrome browser and see what the result of this code looks like.

[Video description begins] *A tab opens in a browser. On the left, a section titled Index of/Project/React/ListsAndKeys/ListsWithKey/ is present. It has two links: ListsKey.html and ListsKey.js. On the right, three tabs are present: Elements, Console, and Sources. Console is currently selected.* [Video description ends]

Click ListsKey.html. The list is rendered as before, but there is a big red warning.

[Video description begins] *The following result appears on the left: Esther, Lily, Candace, Edward, and Sidney. On the right, a warning appears.* [Video description ends]

The warning that you see here on screen is actually a different one from what we encountered earlier. And it's actually more specific, encountered two children with the same key, 1. Key should be unique. This warning also gives us more information about how React uses these keys. Keys should be unique so that components maintain their identity across updates. Having a virtual DOM for efficient re-endering of only those components that have been updated requires that sibling elements, that is, elements at the same level, have keys for React to be able to uniquely identify those elements.

When all of the keys are the same, the behavior is undefined and may change in later versions, so you might get something unexpected. If you expand this warning, you'll be able to view a stack trace of where exactly this warning originated. You can see that it's in the react-dom.development API. And here at the bottom, you can see that the original warning was in babel.js. Clearly, having the same key for each list item is not a feasible option. So let's go back to our code editor, and instead of having key be 1 for each element, I'm going to have the key be name, itself.

[Video description begins] *She shifts back to the tab titled ListsKey.js. She modifies code line 6. It now reads: (name) => <li key={name}>{name.toUpperCase()}</li>.* [Video description ends]

So the key is equal to the content of this list item that we're about to display. The content will be in upper case, the key is the name in whatever case was originally specified. Is this a good idea? Well, what does React need from the keys? Keys should be present for elements which are at the same level, and keys should be unique.

There is no way we can guarantee that the content displayed in the each of the list items is actually unique. So if you have two of the same names in a list of names, then you'll have a key with duplicates. Right now, our list here contains all unique names. But what if there were two entries in the list with the same name? What if the name Candace was present twice?

[Video description begins] *She modifies code line 16. It now reads: const names = ['Esther', 'Lily', 'Candace', 'Candace', 'Edward', 'Sidney'];.* [Video description ends]

What would happen to this list here? Let's go back and hit refresh after having saved our file. And once again, we get the same warning, encountered two children with the same key, Candace.

[Video description begins] *She shifts back to the browser.* [Video description ends]

You cannot rely on user-provided content to be unique, so you can't use them to be the keys of your list items or any elements which are at the same level. You have to generate your own key, which is guaranteed to be unique. So a fix for this problem that would work, and which is much better than any of the options that we've tried so far, is to have an index associated with each item in the list.

[Video description begins] *She shifts back to the tab titled ListsKey.js.* [Video description ends]

Every item in the list will have a unique index. The index is basically the position of that item in the list. How do we apply this index? Well, the function that you pass into the map function can take in two input arguments. The first is the element of the list on which we want to operate, represented by the name variable. The second is the index of that element in the list.

Simply expand your arrow function to take in this additional input argument, and that value will be passed in. The index value will be available within your function, and instead of key is equal to name, change the key to be equal to index. With this change made, we can be sure that the key associated with each list item will be unique, allowing React to uniquely identify each element.

[Video description begins] *She modifies code line 6. It now reads: (name, index) => <li key={index}>{name.toUpperCase()}</li>.* [Video description ends]

Let me just warn you, though, that this code is not perfect. There are still a few issues with this, which we'll fix in the next demo. The first issue that still exists in our code is that using the index is still fairly fragile. It's not a robust option. The second issue is that we've associated the key with this li element. The key should be associated with a higher-level component, if possible. Notwithstanding these issues, what we have still works, and it's better than the options that we've tried so far. Save this file, and head back to our Chrome browser and hit Refresh.

[Video description begins] *She shifts back to the browser.* [Video description ends]

You can see that the error has disappeared and all of our list items, even with duplicate content, have been successfully rendered onto the screen. Let's inspect this, and you can see that the HTML structure shows us all of the li elements.

[Video description begins] *In the left, she clicks on Esther. A context menu appears.She clicks on Inspect. On the right, Elements tab get selected. It has some lines of code. She highlights 6 code lines. Line 1 is: <li>ESTHER</li>== $0. Line 2 is: <li>LILY</li>. Line 3 is:< li>CANDACE</li>. Line 4 is: < li>CANDACE</li>. Line 5 is: < li>EDWARD</li>. Line 6 is:<li>SIDNEY</li>.* [Video description ends]

The key is not rendered in the HTML, though. Remember, the key is a attribute that React knows about, it won't be part of your HTML. We saw that using the index of an element in the list as a key works, but this is a fragile option. Do not use indexes as keys if the list that you're displaying to screen may be modified by the user or behind the scenes. Now, this key is what React uses to check to see whether a particular element has been modified in order to update that element.

Let's say you were to go ahead and insert something in a list as you're displaying it because of some user interaction, then the keys associated with each element might be different, but React may not realize this and end up updating the wrong element. This can cause a very subtle bug which is then hard to track down. React may also think that an element has not been updated because it continues to have the same key as before, but the data may have changed. This can cause all kinds of hair-raising issues, so please don't use indexes as keys if your list might change.

**Correct Key Usage**

[Video description begins] *Topic title: Correct Key Usage. Your host for this session is Janani Ravi.* [Video description ends]

In the last demo, we discussed that even though the index of an element in the list will provide a way to uniquely identify an element, provided the list does not change, using indexes as keys is a fragile solution.

[Video description begins] *A Sublime text editor opens. In the Folders pane, the ExtractingComponents directory is open. In the Editor pane, there are two tabs: ExtractingComponentsWithKeys.html and ExtractingComponentsWithKeys.js.* [Video description ends]

So what is the characteristic that you're looking for in the key that you choose for react elements? You want every item in your list to have a key, which is a permanent and unique property of that item. If there is some unique identifier for that item that you're going to display, that unique identifier should be used as a key. And that's exactly what we'll do here in this demo. Let's switch over to extracting components with keys.js.

Now let's explore this code, starting from the bottom. When you're working in a real world application, when you're looking to display a list on screen, that list generally has some meaning. Maybe it's a list of students, a list of employees. Once you have an entity, you probably have a way to uniquely identify that entity. Take a look at the names constant that I've set up here at the bottom. Instead of using a list, I've specified it in the form of a json object, key value pairs.

Essentially what I've done here is set up a unique identifier for each name. id1 for Esther, id2 for Lily, id3 for Candace, id4 for Edward, and id5 for Sidney. Next, let's take a look at the names list component, which extends the base react component within its render function. We now iterate over each item in our names object in order to display the name to screen. In order to do this, we use the object.keys function available in JavaScript. This will extract each key in a JavaScript object.

The object from which we want to extract keys is this.props.names. We pass in this names object as a part of the props for names list. Use object.keys to extract the keys in that object. That is id1, 2, 3, 4, and 5. Object.keys will extract the keys of an object and return the keys to you in the form of a JavaScript list. We then call the map function on this list, and pass in a function to transform each element in the object. The transformation function takes as its input arguments key as well as index.

And for each element in the object, we render the name component. The props that we pass into the name component is id is equal to key. And name is equal to the value associated with the key. The name component will get two attributes in its props. id, as well as name, and it'll use this information to render out a list item. The names list will simply be an unordered list of these list items as specified by the ul tag here.

[Video description begins] *She highlights code line 15. It reads: <ul className = "names">{listItems}</ul>.* [Video description ends]

Another change here from the previous demo is instead of displaying a list item directly with the name, we've extracted a higher level component, the name component here. And this name component is what we'll look at next.

[Video description begins] *She highlights code line 1. It reads: class Name extends React.Component {.* [Video description ends]

Remember, it has two props that are passed in, id as well as name. It simply renders out a list item element, where the key is this.props.id. And the content of the list item is what we get from this.props.name and we convert it to uppercase. The big improvement here over all of the previous variations of key that we use is that here, the key is a permanent and unique identifier for each name. I'll now use ReactDOM.render, and render the names list out.

And names is the props that I pass in here, which is basically the names object that we have set up in the form of a constant. Save this file and let's head over to our HTTP server and click through to extracting components with keys.html.

[Video description begins] *Five names appear on the page: Esther, Lily, Candace, Edward, and Sidney. A Console pane is open on the right.* [Video description ends]

And here is that warning again. Each child in an array or iterator should have a unique key. But I thought I did provide a unique key. What's wrong? Well, we haven't fully done what React expects us to do. That's because we have a higher level component, the name component that we extracted to wrap around the list item. But we specified the key associated with the li element, rather than the higher level name component.

[Video description begins] *She shifts back to the Sublime text editor.* [Video description ends]

Here is the code that we just ran that caused an error. The key attribute we set on the li html tag. But this li html tag is composed within a higher level component, that is the name component. The correct way to specify the key is to associate it with this higher level name component, not the li tag within this higher level component. Let's make that change and things will become clearer. So I have this name component where I pass in a prop id. I'm going to change this to be the key attribute.

[Video description begins] *She edits code line 12. It now reads: (key, index) => <Name key={key} name={names[key]} />.* [Video description ends]

The key attribute is now associated with the name component, which is a component that we have extracted, which is the component that we display within our list. I'm going to get rid of this key attribute here within the li element. That is the wrong place to specify the key.

[Video description begins] *She edits code line 4. It now reads: <li>{this.props.name.toUpperCase()}</li>.* [Video description ends]

Once we've made these changes, let's go back to where we have this HTML page running on our HTTP server. I'm going to hit refresh and see whether this warning goes away. And yes, indeed it does. So if you've extracted a higher level component which forms the element of your list. That is, which is at the same level, a repeated component at the same level, associate the key with this higher level component.

[Video description begins] *In the right pane, she shifts to the Elements tab.* [Video description ends]

This is the correct usage of the key property in React. Now, this might seem a little confusing, so let me give you a good rule of thumb when you're specifying keys for your components.

[Video description begins] *She shifts back to the Sublime text editor.* [Video description ends]

Anytime you have a map function that you use to display the elements in a list. Here our map function is within names list. Whatever component you render within the map function should have the key attribute associated with it. This component could be a simple HTML tag. If it's a higher level component, the key attribute should be associated with the higher level component. As is the case here, the key attribute should be associated with the higher level name component.

**Rendering Using Conditional If**

[Video description begins] *Topic title: Rendering Using Conditional If. Your host for this session is Janani Ravi.* [Video description ends]

When you work with the React library, you're writing code using component-driven development. Based on the user interface that you want to display for the particular state of your application, you'll devise a component representing that state.

[Video description begins] *A Sublime text editor opens. In the Folders pane, the ConditionalRendering directory is open. In the Editor pane, there are two tabs: ConditionalIf.html and ConditionalIf.js.* [Video description ends]

You'll have distinct components that encapsulate the behavior that you're looking for. And then you can use conditional rendering to render only some of these components, depending on the state of your application. Conditional rendering is extremely powerful because you can use logic constructs to determine what you want to render to screen. You're manipulating your components using logic. We'll start off in this demo by seeing how the conditional, if, works.

Let's switch over to the JS file, because that's where the core of the code is. ConditionalIf.js is the file that we'll use. The first component that I'll set up here is a component that'll display a circle to screen. It's a function component called Circle. It takes in an input argument that is the props and then displays a circle using the background color specified using props.color.

[Video description begins] *She highlights code line 2-4. Code line 2 is: const cirStyle = {. Code line 3 is: backgroundColor: props.color. Code line 4 is: };.* [Video description ends]

The div, which is the React element associated with this component, will display a circle with the style cirStyle.

[Video description begins] *She highlights code lines 6 and 7. Code line 6 is: return <div className="circle". Code line 7 is: style={cirStyle}>.* [Video description ends]

I'm now going to set up another component here called ToggleCircle, which will toggle the color of the circle that is displayed. This is going to be a stateful component. The only bit of state that it holds is a value for the property, go. go is initially initialized to true. The ToggleCircle component will display a circle using red and green colors.

[Video description begins] *She highlights code line 11. It reads: class ToggleCircle extends React.Component {.* [Video description ends]

When the state go is set to true, the circle will be in the green color, indicating a green light, or a green signal.

[Video description begins] *She highlights code line 14.* [Video description ends]

When the state go is equal to false, the circle will be in the red color, indicating a red signal or red light. We'll toggle the state of this component using this toggle method, here.

[Video description begins] *She highlights code line 17.* [Video description ends]

toggle method, defined using the arrow notation, will simply call this.setState.

[Video description begins] *She highlights code line 18. It reads: this.setState(function(prevState) {.* [Video description ends]

We'll access the previous prevState.go and simply toggle the state. If the previous state.go was true, the current state will be false. And if the previous state was false, the current state will set to true.

[Video description begins] *She highlights code line 20. It reads: go: !prevState.go.* [Video description ends]

Based on the current state, whether go is set to true or false, the render function here will display a different version of our component.

[Video description begins] *She highlights code line 25.* [Video description ends]

If this.state.go is equal to true, the code within the if block will be executed.

[Video description begins] *She highlights code line 27. She highlights code lines 28-35. Code line 28 is: return (. Code line 29 is: <div>. Code line 30 is: <Circle color='green' />. Code line 31 is: <button className="button" onClick={this.toggle}>. Code line 32 is: Stop. Code line 33 is: </button>. Code line 34 is: </div>. Code line 35 is: );.* [Video description ends]

And the React element that we create in there is what will be returned. We'll set up a circle with the color green, along with a button that says Stop. The Stop button will allow us to toggle our state from go to stop. The Stop button is wired up to have its onClick handler call this.toggle. That will change the state of this ToggleCircle component. This is the conditional if rendering. If this.state.go is not true, we'll fall through outside this if block, where the circle that we return will have the color red and the button that we'll display will have Go written on it.

[Video description begins] *She highlights code lines 37-44. Code line 37 is: return (. Code line 38 is: <div>. Code line 39 is: <Circle color='red' />. Code line 40 is: <button className="button" onClick={this.toggle}>. Code line 41 is: Go. Code line 42 is: </button>. Code line 43 is: </div>. Code line 44 is: );.* [Video description ends]

This button is wired up to the onClick handler, this.toggle, as well. Based on the current state, the ToggleCircle component will choose to render a different configuration of the Circle component. Now that we've understood how this conditional if rendering works, let's go on and use ReactDOM.render to render this ToggleCircle out to our DOM.

[Video description begins] *She highlights code lines 49 and 50. Code line 49 is: ReactDOM.render(<ToggleCircle/>,. Code line 50 is: document.getElementById('react-conditional-if'));.* [Video description ends]

Switch over to your Chrome browser where you have the HTTP server running, and let's click through to ConditionalIf.html.

[Video description begins] *A page titled Index of/Project/React/ConditionalRendering/ConditionalIf/ is open in the browser. It has two links: ConditionalIf.html and ConditionalIf.js. On the left, the html tool is present. Here, the Console tab is currently open.* [Video description ends]

Remember, the initial state of the component was go set to true, and that's why we have this green circle.

[Video description begins] *A green circle appears on the page. A Stop button is present beneath it.* [Video description ends]

If you can't see the differences in the colors, you can always right-click and inspect the HTML of this component and see what CSS class has been applied, what styles have been applied. Because the current state is go, the button displayed along with the circle says, Stop. If we click on this button, we'll be able to toggle the state to Stop, and that's exactly what I'm going to do here. Click on the Stop button, and you'll see that the circle will be immediately updated to be red in color.

[Video description begins] *A Go button appears in place of the Stop button.* [Video description ends]

The current state was toggled to set go to false, which means the component rendered has also changed. Let's go ahead and click on the Go button, here, and this will toggle the state once again, and once again the green circle is displayed. You can go ahead and try this a couple of times, and you'll see that our conditional rendering works just fine. As the state is toggled, a different component is rendered.

**Conditional Rendering Using Variables**

[Video description begins] *Topic title: Conditional Rendering Using Variables. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll perform conditional rendering using the conditional if as we did before. But instead of rendering components directly, we'll store components in variables. And then render out those variables. So you can see that the code will be cleaner and easier to maintain.

[Video description begins] *A Sublime text editor opens. In the Folders pane, the ConditionalIfWithVariable directory is open. In the Editor pane, there are two tabs: ConditionalIfVariable.html and ConditionalIfVariable.js.* [Video description ends]

Let's head over to our JS code because that's where all of the changes will be, ConditionalIfVariable.js. Once again we'll work with components that we are familiar with. I have a function component called circle which takes as an input argument props, and displays a circle of a certain color.

[Video description begins] *She highlights code line 1.* [Video description ends]

I set the color of the circle using the style property on the div.

[Video description begins] *She highlights code lines 6-8. Code line 6 is: return <div className="circle". Code line 7 is: style={cirStyle}>. Code line 8 is: </div>.* [Video description ends]

The background color is available using props.color.

[Video description begins] *She highlights code line 3.* [Video description ends]

The color for the circle will be passed in as a part of the props for the circle. I now have another function component called green circle.

[Video description begins] *She highlights code line 11. It reads: function GreenCircle(props) {.* [Video description ends]

This green circle component will display the circle function component about in the green color. So we invoke the circle component and pass in color ="green".

[Video description begins] *She highlights code line 12. It reads: return <Circle color="green"></Circle>.* [Video description ends]

We have yet another component called RedCircle, which will display the circle in the red color.

[Video description begins] *She highlights code lines 15-17. Code line 15 is: function RedCircle(props) {. Code line 16 reads: return <Circle color="red"></Circle>. Code line 17 is: }.* [Video description ends]

The value passed in for the color props here is red. Now that we have the circle component setup, I'm now going to create components to represent the Stop button and the Go button. The first component is a function component called StopButton, which simply renders a button out to screen.

[Video description begins] *She highlights code line 19. It reads: function StopButton(props) {.* [Video description ends]

It has an onClick handler, which invokes the function in props.toggle. So the toggle function that we'll define in a top level component we'll pass in as a part of the props for this StopButton component.

[Video description begins] *She highlights code line 21. It reads: <button className="button" onClick={props.toggle}>.* [Video description ends]

The GoButton is also a function component, which displays a button with the message Go. The onClick handler for this Go button invokes props.toggle.

[Video description begins] *She highlights code line 27. It reads: function GoButton(props) {.* [Video description ends]

So the toggle function that we'll define the higher level component is passed in as a part of the props for this component.

[Video description begins] *She highlights code line 29. It reads: <button className="button" onClick={props.toggle}>.* [Video description ends]

Now let's go on and set up the ToggleCircle component. The functionality of this component will be similar to the functionality of the ToggleCircle component from the previous demo, but the way we set up the code will be different. ToggleCircle has some initial state, go is initialized to true.

[Video description begins] *She highlights code line 35. It reads: class ToggleCircle extends React.Component {.* [Video description ends]

We have the toggle function that changes the state of the component.

[Video description begins] *She highlights code lines 37-39.* [Video description ends]

Within the toggle function we invoke the setState function, and we access the previous value of go from the previous state input argument and flip its value. So if the previous value was true, the current value will be false. If the previous value was false, the current value will be true.

[Video description begins] *She highlights code lines 42-45. Code line 42 is: this.setState(function(prevState) {. Code line 43 is: return {. Code line 44 is: go: !prevState.go. Code line 45 is: }.* [Video description ends]

Now that we have these individual components setup, the render function of this ToggleCircle is where we'll see major code changes. Within the render functions, I'm going to set up two variables called button and circle.

[Video description begins] *She highlights code line 50. It reads: let button, circle;.* [Video description ends]

button and circle will be assigned the right component values based on the conditional if check that we perform.

[Video description begins] *She highlights code lines 52-58. Code line 52 is: if (this.state.go) {. Code line 53 is: circle = <GreenCircle />. Code line 54 is: button = <StopButton toggle={this.toggle} />. Code line 55 is: } else {. Code line 56 is: circle = <RedCircle />. Code line 57 is: button = <GoButton toggle={this.toggle} />. Code line 58 is }.* [Video description ends]

If this.state.go is equal to true, we'll enter into the if block. Here the circle displayed will be the green circle, and the button that we'll display will be the stop button. The stop button needs a single value for props that needs to be passed in, and that is the toggle function, this.toggle. If this.state.go is false, we'll execute the code present in the else block.

Here, the circle is initialized to the RedCircle, that's what will be displayed, and the button is initialized to the GoButton. The GoButton also needs access to the toggle event handler which is passed in as a part of its props. Toggle is equal to this.toggle. And finally, the React element that is returned from this render function uses the variables that we had initialized earlier, circle, and then button.

[Video description begins] *She highlights code lines 60-65.* [Video description ends]

These variables are expressions that need to be evaluated, so they need to be specified within curly braces. All that's left to do here is to invoke ReactDOM.render and render to our DOM the ToggleCircle component.

[Video description begins] *She highlights code lines 69 and 70. Code line 69 is: ReactDOM.render(<ToggleCircle/>,. Code line 70 is: document.getElementById('react-conditional-if'));.* [Video description ends]

Make sure you save this file and let's switch over to our HTTP server and our Chrome browser, and take a look at the ConditionalIfVariable.html file.

[Video description begins] *A page titled Index of/Project/React/ConditionalRendering/ConditionalIfWithVariable/ is open in the browser. It has two links: ConditionalIfVariable.html and ConditionalIfVariable.js. On the left, the html tool is present. Here, the Console tab is currently open. A green circle appears on the page. A Stop button is present beneath it.* [Video description ends]

The initial state of the component has Go set to true. That's why we have a green circle and the stop button. If you go ahead and click on the stop button, the state will be toggled. Clicking on stop, we'll switch over to the red circle and the Go button and you can try this over and over again. Every click of the button will toggle the state.

**Conditional Rendering Using Inline Logical Operators**

[Video description begins] *Topic title: Conditional Rendering Using Inline Logical Operators. Your host for this session is Janani Ravi.* [Video description ends]

In the earlier demo, we performed conditional rendering of one kind of component or another based on checks that we performed using either an if statement or an if-else statement.

[Video description begins] *A Sublime text editor opens. In the Folders pane, the InlineLogicalOperators directory is open. In the Editor pane, there are two tabs: InlineLogicalOperators.html and InlineLogicalOperators.js.* [Video description ends]

You can perform conditional rendering using inline logical operators as well, and that's what we'll study here in this demo. JavaScript supports a bunch of logical operators, such as the && for and, the || for or, and the !, or the bang, for not. And all of these can be used to conditionally render the right component of your choice. Here I am within the folder, InlineLogicalOperators, I have a subfolder here called images. And if you expand this subfolder, you’ll find that there are two images here which we'll use in our demo.

The ok image and the warning image. The ok image shows a thumbs up. And the warning image has an image which indicates a warning. The HTML file for this demo is the InlineLogicalOperators.html file. As usual, there's just some CSS classes here which you can check out at leisure. We'll head over to InlineLogicalOperators.js. There, we'll write the JavaScript code for our React components. The first component here is a class that displays a warning.

[Video description begins] *She highlights code line 1.* [Video description ends]

Class Warning extends the base React.Component class, and it has just a render function. It's a stateless component. Within the render function, we display the image, images/warning.png. And we also display some text, the text simply says, something wasn't right, you might need to debug.

[Video description begins] *She highlights code line 6.* [Video description ends]

[Video description begins] *She highlights code line 9.* [Video description ends]

This is a component that displays a generic warning to the user. You can't configure this component, you can't customize your warning. Let's now set up another component called the Ok component, which tells the user that everything is okay.

[Video description begins] *She highlights code line 16. It reads: class Ok extends React.Component {.* [Video description ends]

This Ok component also displays an image, the image source is in images/ok.png, and it also displays some text. The text says, everything looks good, please carry on.

[Video description begins] *She highlights code lines 21 and 24.* [Video description ends]

So it's a very generic Ok component telling the user that all is well. Once again, this is a stateless component and it also doesn't accept any props, which means you can't customize this component in any way. Let's scroll down a little further and write some more code. The Display component here displays either a warning or ok based on whether a warning occurred or not. And for this, we'll use an inline logical operator.

[Video description begins] *She highlights code line 31. It reads: class Display extends React.Component {.* [Video description ends]

The Display component first displays a header, which is an h1 header that says Hello. And after the header, we have an expression which checks to see whether this.props.warning is true.

[Video description begins] *She highlights code line 37.* [Video description ends]

One of the props that we pass into this Display component is the property warning. Which is set to true when a warning needs to be displayed, and which is set to false when no warning needs to be displayed and we have to show the ok dialogue.

[Video description begins] *She highlights code line 39.* [Video description ends]

We use the inline logical and and the inline logical or operator to determine the right component to render. We have a large expression in curly braces, and within the curly braces we have two expressions separated by the logical or. Both of the expressions are in parentheses. The first expression in parentheses checks to see whether this.props.warning and Warning, which means if this.props.warning is true, the warning component will be rendered. If the first expression in parentheses is false, we'll switch over to the other expression after the or sign, or Ok.

This basically means if warning is true, then the Warning component will be rendered. Or if warning is false, the Ok component will be rendered. Let's try the first rendering of this component with the warning props set to true. So ReactDOM.render, the Display component warning is equal to true, which means the first expression in parentheses. (this.props.warning && <Warning />) should evaluate to true.

[Video description begins] *She highlights code line 46. It reads: ReactDOM.render(<Displaywarning={true}/>, document.getElementById('outer'));.* [Video description ends]

Save this file and switch over to your Chrome browser, where we have our HTTP server running. Go to InlineLogicalOperators.html, and immediately you can see that the warning is shown.

[Video description begins] *A page titled Index of/Project/React/ConditionalRendering/InlineLogicalOperators/ is open in the browser. It has the following links: images, DS Store, InlineLogicalOperators.html, and InlineLogicalOperators.js. On the left, the html tool is present. Here, the Console tab is currently open.* [Video description ends]

Something wasn’t right, you might need to debug. Because warning was set to true, the first expression evaluated to true and the Warning component was rendered. I’m now going to set a warning to be equal to false.

[Video description begins] *She shifts to the Sublime text editor.* [Video description ends]

Now if you save these changes and switch over and hit Refresh on Chrome browser, you'll see that the Ok component is rendered. Everything looks good, please carry on.

**Ternary Operators and Preventing Rendering**

[Video description begins] *Topic title: Ternary Operators and Preventing Rendering. Your host for this session is Janani Ravi.* [Video description ends]

In this demo, we'll see how we can use a ternary operator to determine which component to render.

[Video description begins] *A Sublime text editor opens. The editor pane has two tabs: PreventingRendering.html and PreventingRendering.js. The PreventingRendering.html tab is currently active. It has some code lines.* [Video description ends]

The ternary operator, you might already know, is the equivalent of an if-else conditional check. Based on the current state of your component, or the props that have been passed into a component, it's quite possible that under certain conditions you don't want a component to display at all. In this demo, we'll also see how you can prevent a component from rendering. This HTML file that we'll use for this demo references the PreventingRendering.js file, which is where we'll write our code. Head over to this JS file, and let's take a look at our first component here, the Warning component.

[Video description begins] *She highlights code line 1. It reads: class Warning extends React.Component {.* [Video description ends]

This Warning component is similar to the Warning component that we had in the previous demo, except for one change. Let's first take a look at what is similar. The React element returned by this Warning component is the same. There is an image source which points to images/warning.png.

[Video description begins] *She highlights code line 10. It reads: <img src="images/warning.png"/>.* [Video description ends]

And there is some text that is in this warning component. Here is a warning to scare you! That's what this text says.

[Video description begins] *She highlights code lines 12-14. Code line 12 is: <span className="message">. Code line 13 is: Here is a warning to scare you!. Code line 14 is: </span>.* [Video description ends]

But if you move your eyes up to the first few lines of code within this render function, you'll see where we prevent this component from rendering under certain conditions.

[Video description begins] *She highlights code lines 2-5. Code line 2 is: render() {. Code line 3 is: if(!this.props.warn) {. Code line 4 is: return null;. Code line 5 is: }.* [Video description ends]

This Warning component accepts as a part of its props a value for the warn property. If warn is set to true, then the warning component will render. If this.props.warn is equal to false, then the render function for this component will return null. When the React element returned by the render function is null, there is no component to be rendered to screen. The component will display nothing, this is how you prevent rendering. The second component whose code I'm going to paste in right now is the Display component. It extends the base React.Component base class and holds state.

[Video description begins] *A set of code appears in lines 20-43. She points at code lines 20-23. Code line 20 is: class Display extends React.Component {. Code line 21 is: state = {. Code line 22 is: warn: true. Code line 23 is: }.* [Video description ends]

The state object has just one property, warn, which is initially set to true. This Display component also has a button that will allow us to toggle the state of the component.

[Video description begins] *She highlights code lines 36-38. Code line 36 is: <button className="button" onClick={this.toggle}>. Code line 37 is: {this.state.warn ? 'Hide' : 'Show'}. Code line 38 is: </button>.* [Video description ends]

When the button is clicked, this toggle function will be invoked.

[Video description begins] *She highlights code lines 25-31. Code line 25 is: toggle =() => {. Code line 26 is: this.setState(function(prevState) {. Code line 27 is: return {. Code line 28 is: warn: !prevState.warn. Code line 29 is: }. Code line 30 is: });. Code line 31 is: }.* [Video description ends]

Within the toggle function, we'll call this.setState, and basically toggle the state of warn. If prevState.warn was true, the new value for warn will be false. If prevState.warn was false, the new value for warn will be true. This is how we toggle the current state of the component. When the state of the component is toggled, the component is re-rendered. Let's take a look at the render function here. The Display component first displays a button whose onClick handler invokes this.toggle.

[Video description begins] *She highlights code line 36.* [Video description ends]

The contents of the button depends on the value of warn in the current state of the component. And this is where we've used a ternary operator.

[Video description begins] *She highlights code line 37.* [Video description ends]

The ternary operator in JavaScript uses the question mark and the colon. If this.state.warn is set to true, then the button will display the Hide text. If this.state.warn is false, the button will display the text, Show. The Display component then renders the Warning component and passes in a value for the warn props. The value of warn comes from this.state.warn, where the warning should be displayed or not.

[Video description begins] *She highlights code line 39. It reads: <Warning warn={this.state.warn}/>.* [Video description ends]

All that's left to do now is to invoke ReactDOM.render, and render this Display component.

[Video description begins] *She highlights code line 45. It reads: ReactDOM.render(<Display />, document.getElementById('outer'));.* [Video description ends]

We'll then click on the button to toggle the state of the component and see how the warning dialogue is either shown or not. Let's switch over to our Chrome browser and go to PreventingRendering.html.

[Video description begins] *A webpage titled Index of /Project/React/ConditionalRendering/PreventingRendering/ opens. The screen is divided into two parts. On the left, two links are present. She clicks the following link: PreventingRendering.html. A button titled Hide appears. Below it, a warning message is displayed. On the right, there are three tabs: Elements, Console, and Sources. The Console tab is currently open.* [Video description ends]

Initially, warn is set to true, which is why the button displays Hide, and you can see the Warning component, 'Here is a warning to scare you!' Click on this Hide button, here, and this will toggle the state of the component. This sets the warn property to false within the Display component, which means the Warning component will have its render function return null. The component does not render to screen because there is no element to render. this.props.warn will be false and the Warning component render function will return null.

[Video description begins] *She clicks the Hide button. The button name changes to Show. A screenshot of code lines 1-18 appears. She highlights code lines 2-5.* [Video description ends]

If you click on Show, once again, warn will be set to true and the Warning component will reappear. The Warning component no longer returns null, it returns a React element that is rendered to the DOM.

**Local State**

[Video description begins] *Topic title: Local State. Your host for this session is Janani Ravi.* [Video description ends]

As you get more familiar with React and start building up your components in the right way, a question that might arise in your mind is, where exactly should state be stored when you have a component tree?

[Video description begins] *A Sublime text editor opens. In the editor, there are two tabs: LocalState.html and LocalState.js. The LocalState.html tab is currently active. It has some code lines.* [Video description ends]

How do I make the right decision about where state should be held and which components should only receive props? Well, the right answer to this question is, state should be lifted into a higher level component. It should be as high up as possible in your component tree. When you build up your component tree in a React and have different components render different bits of data, it's important that you lift state up to the highest possible component in the component tree.

This will allow you to synchronize your state across multiple components within your app, and that's exactly what we'll see in this demo. In this demo, we'll set up components with local state, we won't lift state up, and you'll see the perils of working with components like this. Once you understand how it is to work with state which is low down in the component tree hierarchy, you'll see how we can lift state up to improve the design of our React components. LocalState.js is the JavaScript file that we'll be working with.

[Video description begins] *She shifts to the LocalState.js tab.* [Video description ends]

We'll be setting up a number of different components here in this demo, and we'll build up the components incrementally.

[Video description begins] *A set of code appears in lines 1-3.* [Video description ends]

I'm going to set up a constant here called currencyNames that will act as an enum.

[Video description begins] *She highlights code line 1. It reads: const currencyNames = {.* [Video description ends]

The d references currency in dollars.

[Video description begins] *She highlights code line 2. It reads: d: 'Dollars'.* [Video description ends]

This is an enum that will be referenced by the components that I set up in just a bit. Next I'll define a component called NetWorthInput, which extends the base React.Component class.

[Video description begins] *A set of code appears in lines 5-14. She highlights code line 5. It reads: class NetWorthInput extends React.Component {.* [Video description ends]

The NetWorthInput component will allow us to enter our net worth in a currency of our choice. To start off with, we'll have this component accept the net worth of an individual in dollars. That is the only currency supported at this point in time. And we'll add in an additional currency later on. The constructor for this NetWorthInput component, it accepts props and calls super props, and it also has local state. We have a state object within this component which holds the net worth of an individual. The state object has a property, local\_networth, which is initialized to 0.

[Video description begins] *She highlights code lines 7-11. Code line 7 is: constructor(props) {. Code line 8 is: super(props);. Code line 10 is: this.state = {. Code line 11 is: local\_networth: 0.* [Video description ends]

We'll have form elements in this component where we'll allow the user to input their net worth.

[Video description begins] *A set of code appears in lines 15-19.* [Video description ends]

The input text box will be wired up to this handleChange event handler.

[Video description begins] *She points at code line 15. It reads: handleChange = (e) => {.* [Video description ends]

This event handler simply updates the local state of this NetWorthInput component. We'll call this.setState and update the local\_networth.

[Video description begins] *She highlights code lines 16-18. Code line 16 is: this.setState({. Code line 17 is: local\_networth: parseFloat(e.target.value) || 0. Code line 18 is: });.* [Video description ends]

We'll parse as a floating point, e.target.value. That is a value from the input element. And if it's a valid floating point value, we'll assign that to the local\_networth, otherwise we'll set local\_networth to 0. Next up is the render function for this component.

[Video description begins] *A set of code appears in lines 21-32.* [Video description ends]

The render function will first access the currency that you want your net worth to be input in. Right now we just support dollars. Keep that in mind but we'll access the currency using this.props.currency.

[Video description begins] *She highlights code line 22. It reads: const currency = this.props.currency;.* [Video description ends]

I've defined a field set here which groups together logically-related components in my form.

[Video description begins] *She highlights code line 24. It reads: <fieldset className = "networth">.* [Video description ends]

Within the field set, I have the legend that says, 'Enter net worth in {currencyNames[currency]}.' So look up the name of the currency using the currency constant specified.

[Video description begins] *She highlights code line 25. It reads: <legend>Enter net worth in {currencyNames[currency]} </legend>.* [Video description ends]

The currency constant passed into this component will have d, for dollars, or some other character for other currencies that we may support in the future.

[Video description begins] *A screenshot of code lines 1-3 appears. She highlights code line 2.* [Video description ends]

Within this field set we now have an input text box where the default value is set to this.state.local\_networth.

[Video description begins] *She highlights code lines 27 and 28. Code line 27 is: <input className='input'. Code line 28 is: defaultValue={this.state.local\_networth}.* [Video description ends]

When you specify the default value attribute for a form element, this is how you set the initial value for that element. Any updates made to this input is not controlled by React, the element handles those updates itself. Any time the user updates the input specified in this text box, the onChange event handler will be invoked, this.handleChange.

[Video description begins] *She highlights code line 29.* [Video description ends]

And handleChange in turn will update the state of the component. It'll call this.setState to update the local\_networth. This is a stateful component, net worth input.

[Video description begins] *She highlights code lines 16 and 17.* [Video description ends]

I'm now going to set up another component, this time this is a function component called Millionaire.

[Video description begins] *A set of code appears in lines 5-11. The succeeding code lines shift eight lines down. She highlights code line 5. It reads: function Millionaire(props) {.* [Video description ends]

This Millionaire component will keep track of the net worth of an individual and will tell you whether this individual is a millionaire or not. The props passed into this function component will contain the net worth of the individual. Observe that we haven't specified whether the net worth is in dollars or some other currency.

[Video description begins] *She highlights code line 6. It reads: if (props.networth > 1000000) {.* [Video description ends]

The millionaire component will simply check the net worth, see if it's over 1 million, if yes, it will return the text, A millionaire, otherwise it'll return the text, Not yet a millionaire.

[Video description begins] *She highlights code lines 7 and 10. Code line 7 is: return <p className = "millionaire">A millionaire!</p>;. Code line 10 is: return <p className = "not-millionaire">Not yet a millionaire...</p>;.* [Video description ends]

The millionaire component only checks for the net worth, whether it's over a million or not, it doesn't worry about the currency. I'm now going to include this millionaire component within my NetWorthInput component.

[Video description begins] *The following code appears in line 39: <Millionaire networth={this.state.local\_networth} />. The succeeding code lines shift a line down.* [Video description ends]

Within this field set we have an input text box where the user can type in his or her net worth. As the net worth is typed in, the local\_networth in the state of this component will be updated. This local\_networth is what we pass into the Millionaire component, and the Millionaire component will check to see whether the user is a millionaire or not. Now this NetWorthInput component, I'm going to use within my component called MillionaireCalculator.

[Video description begins] *A set of code appears in lines 46-52. Code line 46 is: class MillionaireCalculator extends React.Component {. Code line 48 is: constructor(props) {. Code line 49 is: super(props);. Code line 51 is: }. Code line 52 is: }.* [Video description ends]

This is the higher level component that will contain nested within it, the NetWorthInput component that we had set up earlier. The render function of this MillionaireCalculator simply renders out to screen, NetWorthInput.

[Video description begins] *A set of code appears in lines 53-59. She highlights code line 56. It reads: <NetWorthInput currency={'d'}/>.* [Video description ends]

Remember that NetWorthInput needs some prompts to be passed in. We need to specify the currency in which the user has to input his or her net worth. I've set currency equal to d, indicating that our input is going to be in dollars. I'll now render our MillionaireCalculator to the DOM using ReactDOM.render.

[Video description begins] *The following code appears in line 62: ReactDOM.render(< MillionaireCalculator />, document.getElementById('outer'));.* [Video description ends]

This is the MillionaireCalculator where the NetWorthInput has local state. We've stored our state low down in the component hierarchy within NetWorthInput.

[Video description begins] *A webpage titled Index of/Project/React/LiftingStateUp/LocalState opens. The screen is divided into two parts. On the left, two links are present: LocalState.html and LocalState.js. On the right, there are three tabs: Elements, Console, and Sources. The Console tab is currently open.* [Video description ends]

Here we are on Chrome browser with our HTTP server running, click through to LocalState.html, and let's take a look at our form, here.

[Video description begins] *A form titled Enter net worth in Dollars appears on the left. A text field is present. Below it, the following message is displayed: Not yet a millionaire...* [Video description ends]

This is our field set, here, which logically groups together form elements. We have an input text box which allows us to enter our net worth.

[Video description begins] *A screenshot of code lines 21-34 appears. She highlights code lines 27-29.* [Video description ends]

The default value here is this.state.local\_networth, which has been initialized to 0. Below that we have the MillionaireComponent, where we've passed in our local\_networth, which is currently 0. That's why it says, Not yet a millionaire. I'm going to display the elements that make up this page using Chrome Browser Tools and expand the HTML structure. So you can see what this page looks like and how the page changes as we make updates.

[Video description begins] *She shifts to the Elements tab.* [Video description ends]

Now let's go ahead and start inputting some text. I'm going to slowly start typing into this text box here. I start with the number 1, and I'll add in 0s, till at a point in time I'll reach a million dollars, that's my net worth. And immediately you can see the Millionaire component is updated to say, A millionaire! If I get rid of a couple of 0s, my Millionaire component tells me I'm not yet a millionaire. I'll add a few 0s, and once again, it'll say I'm a millionaire. Here we have just one net worth component which accepts our net worth in dollars.

**Disadvantages of Local State**

[Video description begins] *Topic title: Disadvantages of Local State. Your host for this session is Janani Ravi.* [Video description ends]

Let's go back to our code here and add in another net worth component. This time one that accepts our net worth in Rupees.

[Video description begins] *A Sublime text editor opens. A tab titled LocalState.js is currently open in the editor pane. It has some code lines.* [Video description ends]

This will require us to first edit our currency names enum, so that the character r represents the net worth in Rupees.

[Video description begins] *She highlights code lines 1-3. Code line 1 is: const currencyNames = {. Code line 2 is: d: 'Dollars',. Code line 3 is: r: 'Rupees'.* [Video description ends]

Our net worth component can already accept net worth in any currency that we specify. I'm just going to go into the MillionaireCalculator and add in another net worth input component, this time with currency equal to r.

[Video description begins] *In line 58, she adds the following code: <NetWorthInput currency={'r'}/>. The succeeding code lines shift a line down.* [Video description ends]

Save the changes that you made your file and let's head back to the Chrome browser and hit refresh.

[Video description begins] *A webpage titled /Project/React/Lifting StateUp/LocalState/LocalState.html opens. The screen is divided into two parts. On the left, there are two sections: Enter net worth in Dollars and Enter net worth in Rupees. Each section contains an input field along with the following message: Not yet a millionaire. On the right, there are three tabs: Elements, Console, and Sources. The Elements tab is currently open.* [Video description ends]

Now, our HTML page has two net worth input components. One way you can enter your net worth in dollars, and another where you can enter your net worth in rupees. Let's head over to our Chrome developer tools and expand the HTML structure of this page. Here is our first net worth input elements. We have the field set tag with the legend which says, Enter net worth in Dollars. We have the input textbox. And we have the millionaire component which says Not yet a millionaire. That's because our initial net worth has been set to zero. You can expand the field set that represents our second net worth component. You can see the legend here, it says Enter net worth in Rupees.

Our net worth in Rupees is also initialized to 0, our millionaire component says Not yet a millionaire. The problem with local state here is that these components are not synchronized, so I've entered my net worth in Dollars. And I get the message A millionaire. In the top component here, this millionaire refers to a dollar millionaire. I have no idea what my net worth is in rupees. So if I remove a couple of zeroes, it says Not yet a millionaire. Once again, the first component always refers to a dollar millionaire. The second component here that says Enter net worth in Rupees will completely work in Rupees. So I enter my net worth. And this millionaire message here in the second component refers to a millionaire in Rupee terms, which is not the same as a millionaire in dollar terms.

There is a conversion factor that hasn't been taken into account. As I see it, there are two problems here. Both of these components are not synchronized. There is no conversion going on between dollars and rupees, so I can't see my net worth in dollars and rupees at the same time. The second problem here is that one component refers to millionaires in dollar terms, the second component refers to millionaires in rupee terms. It's not the same, 70 rupees make a $1 at the time of this recording. So a dollar millionaire is very different from a rupee millionaire and we need to bring across this different, and we'll do this by lifting state up.

**Removing State from Lower Level Components**

[Video description begins] *Topic title: Removing State from Lower Level Components. Your host for this session is Janani Ravi.* [Video description ends]

In the previous demo, we encountered some of the drawbacks of using state that are local to components, instead of in a higher level component.

[Video description begins] *A Sublime text editor opens. In the editor, there are two tabs: LiftingStateUp.html and LiftingStateUp.js. The LiftingStateUp.html tab is currently active. It has some code lines.* [Video description ends]

We had multiple networth input components. But they were not synchronized because each networth input component held its own local state to store the net worth. Each component, which represented the net worth in a different currency, denoted millionaires based on that currencies. So there was no common definition for a millionaire, a dollar millionaire or a rupee millionaire.

So this is highly confusing when you set up components which are related to each other but they have local state. What you need to do to fix this is to lift state up, and that's exactly what we'll do here in this demo. LiftingStateUp.html, all of our interesting code will be in LiftingStateUp.js. So let's head over there. In this demo, we'll work with the same components that we saw in the previous demo, the NetWorthInput component which allows you to input your net worth using different currencies. The two currencies that we'll work with is dollars and Indian rupees.

[Video description begins] *Four code lines appear in lines 1-4. Code line 1 is: const currencyNames = {. Code line 2 is: d: 'Dollars'. Code line 3 is: r: 'Rupees'. Code line 4 is: };.* [Video description ends]

Here is a JavaScript object currencyNames that serves as an enum to represent this information, d for dollars and r for Rupees. Now, there is obviously an exchange rate to convert between these two currencies.

[Video description begins] *A set of code lines appears from lines 6-12.* [Video description ends]

I want the components to display the same information, I want them to be synchronized. So if you input your net worth in dollars, you should see your corresponding net worth in rupees and vice versa. In order to perform the synchronization, I'll need to perform the exchange rate conversion. And I'll do so at whatever exchange rate exists at the time of this recording. This may not, of course, hold in the future. I have a function here called toDollars where you specify your net worth in rupees, that is the input argument. And it converts it to the corresponding equivalent in dollars.

[Video description begins] *She highlights code line 6. It reads: function toDollars(rupees) {.* [Video description ends]

Rupees multiplied by 0.014 will give you your net worth in dollars.

[Video description begins] *She highlights code line 7. It reads: return (rupees \* 0.014);.* [Video description ends]

Similarly, I have a function called toRupees which accepts your net worth in dollars and converts it to rupees.

[Video description begins] *She highlights code line 10. It reads: function toRupees(dollars) {.* [Video description ends]

Dollars multiplied by 70.95 gives you your net worth in rupees.

[Video description begins] *She highlights code line 11. It reads: return (dollars \* 70.95);.* [Video description ends]

The exchange rate that I have considered here is roughly 71 rupees to a dollar. Once I have these helper functions that help me convert from dollars to rupees and vice versa, I'm now going to set up yet another helper function to actually perform this conversion.

[Video description begins] *A set of code lines appears from lines 14-25.* [Video description ends]

This is the tryConvert function which takes in two input arguments, money and convert.

[Video description begins] *She highlights code line 14. It reads: function tryConvert(money, convert) {.* [Video description ends]

Money is a numeric value, which is just the net worth specified either in dollars or in rupees. Convert is the conversion function which will perform the actual conversion based on the currency that you use to specify the money value. It will go from either dollars to rupees or rupees to dollars. Within the tryConvert function, we'll first convert the money input argument to a floating point number using parseFloat. We'll save this in the constant input.

[Video description begins] *She highlights code line 15. It reads: const input = parseFloat(money);.* [Video description ends]

We'll then check to see whether input is a valid number or not. If it's a NaN, that is Number.isNaN is true, we'll simply return an empty string. We were unable to perform the conversion.

[Video description begins] *She highlights code lines 17 and 18. Code line 17 is: if (Number.isNaN(input)) {. Code line 18 is: return '';.* [Video description ends]

However, if input is a valid floating point number, we'll pass that as an input argument to the convert function, and we'll get a constant that we'll store in output.

[Video description begins] *She highlights code line 21. It reads: const output = convert(input);.* [Video description ends]

We'll then round the converted output to two decimal places, and then return the rounded output in the form of a string.

[Video description begins] *She highlights code line 22. It reads: const rounded = Math.round(output \* 100) / 100;.* [Video description ends]

[Video description begins] *She highlights code line 24. It reads: return rounded.toString();.* [Video description ends]

This completes our conversion function, so where we convert dollars to rupees and vice versa, rounded off to two decimal places. Let's move on, and this time, we'll set up our millionaire component, which will display only dollar millionaires.

[Video description begins] *A set of code lines appears from lines 27-33. She highlights code line 27. It reads: function Millionaire(props) {.* [Video description ends]

Note that I'm explicitly changing how the millionaire component works. It accepts props as its input argument and checks for props.dollars. So you only specify your net worth in dollars to this millionaire component.

[Video description begins] *She highlights code line 28. It reads: if (props.dollars > 1000000) {.* [Video description ends]

If props.dollars is greater than 1 million, we'll print out A dollar millionaire.

[Video description begins] *She highlights code line 29. It reads: return <p className = "millionaire">A dollar millionaire!</p >;.* [Video description ends]

If not, we'll print out Not yet a dollar millionaire.

[Video description begins] *She highlights code line 32. It reads: return <p className = "not-millionaire">Not yet a dollar millionaire...!</p >;.* [Video description ends]

We've made the text in the millionaire component very explicit. We're only checking to see whether someone is a dollar millionaire or not. It's now time for our NetWorthInput component. We're going to be tweaking this component so that it no longer holds local state.

[Video description begins] *A set of code lines appears from lines 35-40.* [Video description ends]

All of the state will be held in a higher level component in our component tree. NetWorthInput extends React.Component.

[Video description begins] *She highlights code line 35. It reads: class NetWorthInput extends React.Component {.* [Video description ends]

And we have a handleChange event handler here where all we do is delegate change to the net worth to this.props.onNetWorthChange. The change value is available in e.target.value.

[Video description begins] *She highlights code lines 37 and 38. Code line 37 is: handleChange = (e) => {. Code line 38 is: this.props.onNetWorthChange(e.target.value);.* [Video description ends]

e.target contains a reference to our form element where we input our net worth. Because the NetWorthInput no longer contains a local state, no local state is updated here. Instead, we simply invoke a function that has been passed in by a higher level component, which will then change the state in that higher level component. Let's move on to the render function of this net worth input component. Remember, this is now a stateless component. So all of the data that it has to display, it will receive via its props.

[Video description begins] *A set of code lines appears from lines 41-49. She highlights code line 42. It reads: const networth = this.props.networth;.* [Video description ends]

this.props.networth contains the value for net worth passed in via props. Now, this net worth could be in any currency. And the currency that was chosen to represent this net worth is available in this.props.currency.

[Video description begins] *She highlights code line 43. It reads: const currency = this.props.currency;.* [Video description ends]

We save both of these into constants, networth and currency. The React element that represents this component is a fieldset as before which logically groups together form elements.

[Video description begins] *A set of code lines appears from lines 47-54. The succeeding code lines shift seven lines down.* [Video description ends]

This fieldset has a legend where we say Enter net worth in and the currency here we look up using the currencyNames enum that we had specified earlier.

[Video description begins] *She highlights code line 48. It reads: <legend>Enter net worth in {currencyNames[currency]}:</legend>.* [Video description ends]

If the currency constant said d, the currency would be dollars, if it said r, the currency would be rupees. We then have our input textbox, where the default value, that is the initial value that React wants to show in this text box, is in the networth constant, passed in via props.

[Video description begins] *She highlights code line 51. It reads: defaultValue={networth}.* [Video description ends]

Whenever the user edits or makes changes to this input textbox, the handleChange event handler will be called.

[Video description begins] *She highlights code line 52. It reads: onChange={this.handleChange} />.* [Video description ends]

handleChange, as you can see at the top of your screen, simply delegates state changes to a higher level component. It just calls this.props.onNetworthChange, a function passed in by a higher level component.

[Video description begins] *She highlights code line 38.* [Video description ends]

**Lifting State up to Higher Level Components**

[Video description begins] *Topic title: Lifting State up to Higher Level Components. Your host for this session is Janani Ravi.* [Video description ends]

We're now ready to setup the MillionaireCalculator component, this is the highest component in our component tree. This is the component what will hold state and pass the state information in the form of props to net worth input.

[Video description begins] *A Sublime text editor opens. A tab titled LiftingStateUp.js is currently open in the editor. It has some code lines.* [Video description ends]

Within the constructor of this component, we initialize the state for this component, networth is set to the empty string.

[Video description begins] *She highlights code lines 64-66. Code line 64 is: this.state = {. Code line 65 is: networth: '',. Code line 66 is: currency: 'd'.* [Video description ends]

Nothing has been input yet, and we set the default currency to be d, that is in dollars. This MillionaireCalculator will allow you to input your net worth either in dollars or in rupees which means we need an event handler to handle input in dollars and input in rupees.

[Video description begins] *A set of code lines appears from lines 70-76.* [Video description ends]

The first event handler handles the networth change when you specify your net worth in dollars. handleDollarChange will accept the input argument networth specified in dollars and update the current state of this React component to set the currency to d, indicating that this is a dollar specification, and the networth to the networth that you've passed in.

[Video description begins] *She highlights code lines 70-72. Code line 70 is: handleDoIIerChange = (networth) => {. Code line 71 is: this.setState({currency: 'd', networth: networth});. Code line 72 is: }.* [Video description ends]

Similarly, handleRupeesChange will handle any changes to the networth input that you specify in rupees. The networth input argument here accepts the networth in rupees and updates the current state of the React component to set currency to r and networth to whatever networth you specified.

[Video description begins] *She highlights code lines 74-76. Code line 74 is: handleRupeesChange = (networth) => {. Code line 75 is: this.setState({currency: 'r', networth: networth});. Code line 72 is: }.* [Video description ends]

With the event handler set up, we can now move on to the render method for this component.

[Video description begins] *A set of code lines appears from lines 78-89.* [Video description ends]

We extract the currency and the networth present in the state of this component, this.state.currency and this.state.networth, and store it in constants.

[Video description begins] *She highlights code lines 80 and 81. Code line 80 is: const currency = this.state.currency;. Code line 81 is: const networth = this.state.networth;.* [Video description ends]

Next, we'll calculate the networth in dollars to store in the const dollars. If the currency was specified in rupees, that is, the user entered their net worth in rupees, currency triple equal to r will be true. We'll then try and convert the networth specified in Rupees toDollars, otherwise we'll store the networth as is in the dollars constant.

[Video description begins] *She highlights code line 83. It reads: const dollars = currency === 'r' ? tryConvert(networth, toDollars) : networth;.* [Video description ends]

Once we've calculated the net worth in dollars, we'll perform a similar calculation to calculate the net worth in rupees to store in the const rupees. We'll check whether the current currency is equal to d, that is the net worth was specified in dollars. If yes, we'll invoke the tryConvert helper function to convert the networth specified in dollars toRupees. If not, we'll accept the net worth as is. It's already in rupees, and the end result will be stored in the constant rupees. At this point in time, we have the net worth both in dollars as well as rupees.

[Video description begins] *She highlights code line 84. It reads: const rupees = currency === 'd' ? tryConvert(networth, toRupees) : networth;.* [Video description ends]

This is important because this is what we're going to use to keep our two net worth input components in sync. One which accepts net worth in dollars and another which accepts net worth in rupees. And with this information available, we can set up the components that we want to display within the MillionaireCalculator.

[Video description begins] *A set of code lines appears from lines 88-99. The succeeding code lines shift twelve lines down.* [Video description ends]

We first have the networth input component, which uses the currency dollars where we show the net worth in dollars.

[Video description begins] *She highlights code lines 89-92. Code line 89 is: <NetWorthInput. Code line 90 is: currency="d". Code line 91 is: network={dollars}. Code line 92 is: onNetWorthChange={this.handleDollarChange} />.* [Video description ends]

Any change to the networth specified in this first networth input component will trigger the handler this.handleDollarChange. We then have the second networth input component, the currency here is rupees, so currency is equal to r.

[Video description begins] *She highlights code line 94. It reads: currency="r".* [Video description ends]

It'll always display the net worth of the individual in rupees.

[Video description begins] *She highlights code line 95. It reads: network={rupees}.* [Video description ends]

If the input net worth in this component is changed, the event handler that will be invoked is handleRupeesChange.

[Video description begins] *She highlights code line 96. It reads: onNetWorthChange={this.handleRupeesChange} />.* [Video description ends]

And finally, at the bottom we have the millionaire component, which will display whether the individual is a dollar millionaire or not.

[Video description begins] *She highlights code lines 97 and 98. Code line 97 is: <Millionaire. Code line 98 is: dollars={parseFloat(dollars)} />.* [Video description ends]

The only input that the millionaire component takes in is the net worth in dollars. It doesn't just accept a net worth number, it explicitly accepts the net worth in dollars so that it can calculate whether the individual is a dollar millionaire or not. Now that we have lifted state up successfully to the MillionaireCalculator component rather than having it stored locally in the networthInput component, let's go ahead and use the ReactDOM.render function to render out the MillionaireCalculator component.

[Video description begins] *The following code appears in line 104: ReactDOM.render(< MillionaireCalculator />, document.getElementById('outer'));.* [Video description ends]

Make sure you save the changes to this file and let's switch over to our Chrome browser and click through to LiftingStateUp.html.

[Video description begins] *A webpage titled Index of/Project/React/LiftingStateUp/LiftingStateUp opens. The screen is divided into two parts. On the left, two links are present. She clicks the following link: LiftingStateUp.html. On the right, there are three tabs: Elements, Console, and Sources. The Console tab is currently open.* [Video description ends]

And here on screen are our two net worth input components, where you can specify your net worth either in dollars or in rupees. We have a single millionaire component which tells you whether you are a dollar millionaire or not. When initially we have everything set to zero, we're not yet a dollar millionaire.

[Video description begins] *A section appears. It has input fields for: Enter net worth in Dollars and Enter net worth in Rupees. Below these, a message is displayed. It reads: Not yet a dollar millionaire.* [Video description ends]

I'm now going to head over to Chrome browser tools to take a look at the HTML structure of this page.

[Video description begins] *She shifts to the Elements tab.* [Video description ends]

I'm going to expand all of the individual elements to see how the components have been rendered into the DOM. You can see that we have two field sets here, one for networth in dollars and another for networth in rupees. In the first component, we'll be able to enter our net worth in dollars, that's what the legend says.

In the second component we'll be able to enter our net worth in Rupees, that's what the legend says here as well. The cool thing here is when I type in my net worth in dollars, an automatic conversion is performed and my net worth is also displayed in rupees at the same time. This is because of how we had set up the render function in the MillionaireCalculator.

[Video description begins] *A screenshot of code lines 78-101 appears.* [Video description ends]

Whatever we input our net worth in, we perform the conversion to the other currency. If we input in dollars, we convert to rupees and vice versa. And we re-render both networthInput components so they are constantly in sync. And my millionaire component will tell me whether I'm a dollar millionaire or not. Once I've gone above a million dollars, I get the message A dollar millionaire! Lifting state up to a higher level component allows me to keep both of these net worth input components in sync. As I specify my net worth in dollars, the rupee component is updated as well.

The conversion is done by a higher level component and both components are updated. I'll now specify my net worth in rupees, and as I type in, my net worth in dollars is displayed simultaneously. My high level component has performed this conversion. And at a point that I become a dollar millionaire, the message changes to tell me, you're a dollar millionaire. Now that you've understood how these components work, I'll leave it to you to play around as you wish. Remember, both components are kept in sync, thanks to our state being held in a higher level component, the MillionaireCalculator.

**Code Reuse Using Inheritance**

[Video description begins] *Topic title: Code Reuse Using Inheritance. Your host for this session is Janani Ravi.* [Video description ends]

In order to have your code in production be maintainable and robust, it's very important that you re-use your code where possible. You shouldn't be duplicating the same functionality in different parts of your code, you should reuse components or objects.

[Video description begins] *A Sublime text editor opens. In the editor, there are two tabs: Inheritance.html and Inheritance.js. The Inheritance.html tab is currently active. It has some code lines.* [Video description ends]

When you're working with React, it's preferable that you use composition instead of inheritance to reuse your code. Now with any object oriented programming language, especially if you've worked in a programming language such as Java, the common practice is to set up a base class with base functionality. And then derived classes for different bits of specific functionality thus reusing the code in the base class. The fact that base class functionality is available in all derived classes make inheritance a common design pattern for code reuse especially if you come from a different programming background such as from Java.

Now in React, it is possible for you to use inheritance to reuse code, but it's not preferred. In this first demo, we'll see how you can set up code reuse using Inheritance, we'll write our code in Inheritance.js. But in later demos we'll improve upon this and see how composition is a better design strategy for code reuse when you're working with React components. Let's head over to inheritance.js, and let's set up a simple component that displays a message to screen.

[Video description begins] *A set of code lines appears from lines 1-15. She highlights code line 1. It reads: class Message extends React.Component {.* [Video description ends]

This is an extremely simple message class it's a stateless component. It's constructor accepts props from a higher level component and we also assign a specific style to this message as well. This.className contains the CSS classes that we'll apply to this component, the border class as well as the plain-message class.

[Video description begins] *She highlights code lines 3 and 5. Code line 3 is: constructor(props) {. Code line 5 is: this.className = 'border plain-message';.* [Video description ends]

Within the render function of this component we'll simply render a div with this CSS className applied and display this.props.message. The message text that has been passed in via props.

[Video description begins] *She highlights code lines 10 and 11. Code line 10 is: <div className={this.className}>. Code line 11 is: {this.props.message}.* [Video description ends]

This is a simple enough component, I'll now set up another component called Dialog which renders this message to screen.

[Video description begins] *A set of code lines appears from lines 17-26. She highlights code line 17. It reads: class Dialog extends React.Component {.* [Video description ends]

Now I've used composition to have the message component be composed within the Dialog component. We haven't yet come to the part where we'll work with inheritance yet. The message component here will display the text, A plain message. The only thing left here is to use ReactDom.render to render this Dialog component to our DOM.

[Video description begins] *She highlights code line 22. It reads: <Message message="A plain message"></Message>.* [Video description ends]

[Video description begins] *The following code appears in line 28: ReactDOM.render(< Dialog />, document.getElementById('react-inheritance'));.* [Video description ends]

Everything is very straightforward, make sure you save the contents of this file and let's head over to the Chrome browser to take a look at our HTML. Click through to inheritance.html and here's what the plain message with a border looks like.

[Video description begins] *A webpage titled Index of/Project/React/Composition/Inheritance opens. The screen is divided into two parts. On the left, two links are present: Inheritance.html and Inheritance.js. On the right, there are three tabs: Elements, Console, and Sources. The Console tab is currently open.* [Video description ends]

[Video description begins] *A text field labelled A plain message appears on the left.* [Video description ends]

Let's take a look at the HTML structure to make sure we have the right CSS classes applied.

[Video description begins] *She shifts to the Elements tab.* [Video description ends]

I'm going to expand the divs within the HTML of my page, and here is my div with the plane message. The class attribute for this div is border and plain-message.

[Video description begins] *She highlights the following code: <div class="border plain-message">A plain message </div>.* [Video description ends]

Let's go back to our editor. Let's say I want to display another kind of message, a success message with a different background color, maybe a different border. I have another class here, SuccessMessage, that extends the base React.Component class.

[Video description begins] *A set of code lines appears from lines 17-31. The succeeding code lines shift sixteen lines down. She highlights code line 17. It reads: class SuccessMessage extends React.Component {.* [Video description ends]

There is no inheritance here at all, I have two different components doing almost exactly the same thing. If you look at the constructor for the SuccessMessage class, it's very similar to the constructor for the message class.

[Video description begins] *She highlights code line 19. It reads: constructor(props) {.* [Video description ends]

this.className is set to border and a slightly different CSS style, success-message is a CSS style.

[Video description begins] *She highlights code line 21. It reads: this.className = 'border success-message';.* [Video description ends]

Within the render function though, the code for SuccessMessage and message is exactly the same. We have a div with the CSS class styles applied, which displays this.props.message.

[Video description begins] *She highlights code lines 26 and 27. Code line 26 is: <div className={this.className}>. Code line 27 is: {this.props.message}.* [Video description ends]

This code here, doesn't involve any reuse at all, and that's pretty bad. Now I'm going to go ahead and paste the SuccessMessage component to have it display below the message component.

[Video description begins] *The following code appears in line 39: <SuccessMessage message="A message indicating success"></SuccessMessage>. The succeeding code lines shift a line down.* [Video description ends]

Save this file and let's switch over to the Chrome browser and hit Refresh. We just have the plain message before, now we have the plain message as well as a message indicating success. We set up two components with almost exactly the same HTML, and you can see this when you expand the HTML structure of your page within Chrome Developer Tools. Having no code reuse at all is pretty bad design. So let's go back to our editor and have the class SuccessMessage. Instead of extending from React.Component we'll have it extend from the message component that we had defined earlier.

[Video description begins] *She edits code line 17. It now reads: class SuccessMessage extends Message {.* [Video description ends]

Inheritance is not the preferred design strategy to use in React, but it does allow for code reuse, so it's better than no code reuse at all. Within the constructor, I realized that the border CSS class applies to both message as well as SuccessMessage. So I'll get rid of this border explicitly and simply use the className specified in the base message class. SuccessMessage thus inherits whatever CSS styles we've applied to the message class, in addition, we add in another CSS style success-message.

[Video description begins] *She edits code line 21. It now reads: this.className = this.className + 'success-message'.* [Video description ends]

And the render function for message as well as SuccessMessage both are very similar. So instead of rewriting this code, I'm simply going to call super.render, and simply use the render function in the superclass that is a message class.

[Video description begins] *She edits code lines 26 and 27. Code line 26 is: <div>. Code line 27 is: {super.render()}.* [Video description ends]

Save these changes, go back to Chrome and hit Refresh. We had a plain message and a message indicating success, but something seems to be wrong here, a message indicating success should have a green background. I must have made a mistake while setting up the inheritance of SuccessMessage from message. I'm going to use Chrome Developer Tools and expand the HTML structure of this page to see what CSS classes have been applied and there is my error. T

he CSS class applied to the SuccessMessage does not have a space between plain message and successmessage. This is something that I'll need to fix in my code, so I'm going to head over to our Sublime Text Editor to add in the space. So I'm going to add in the space just before success message, so that the two class tiles are separated, go back to your Chrome browser and hit Refresh.

[Video description begins] *She modifies code line 21.* [Video description ends]

And this time, a message indicating success will be displayed with the green background. And if you look at the div for the success message in Chrome Developer Tools, you can see that the CSS class have been applied correctly. We have plane message space and then success message. Here we are back on Sublime Text, I'm going to use inheritance for code reuse once again. And setup yet another message, FailureMessage which extends the message base class.

[Video description begins] *A set of code lines appears from lines 33-48. The succeeding code lines shift seventeen lines down. She highlights code line 33. It reads: class FailureMessage extends Message {.* [Video description ends]

If you look at the constructor of this FailureMessage, we reuse this.className from the superclass and append the error message CSS class as well.

[Video description begins] *She highlights code line 38. It reads: this.className = this.className + ' error-message';.* [Video description ends]

And the render function for the FailureMessage simply invokes super.render, because what we are rendering to the screen is exactly the same as in the base message class.

[Video description begins] *She highlights code line 44.* [Video description ends]

Scroll down and add in this FailureMessage as the third component within this Dialog, make sure you save the changes to this file.

[Video description begins] *The following code appears in line 57: <FailureMessage message="A message indicating failure"></FailureMessage>. The succeeding code lines shift a line down.* [Video description ends]

And let's head over to Chrome and hit Refresh. We now have three messages rendered to the screen, a plain message, a message indicating success, and a message indicating failure. And we've applied code reuse using inheritance.

[Video description begins] *The text field labelled A message indicating failure has a red background.* [Video description ends]

You can expand the HTML structure of this page to view the elements corresponding to these three messages.

**Code Reuse Using Composition**

[Video description begins] *Topic title: Code Reuse Using Composition. Your host for this session is Janani Ravi.* [Video description ends]

In the previous demo, we saw how we could use inheritance as a design strategy for code reuse in React component.

[Video description begins] *A Sublime text editor opens. The editor pane has two tabs: Composition.html and Composition.js. Currently, the Composition.html tab is open. It has some code lines.* [Video description ends]

However, if you're working in React, composition is preferred to inheritance. And we'll see how we can set up the same structure as before. This time we'll work with composition rather than inheritance and you'll see how the code is far simpler and easier to read. Let's head over to our JS file composition.js where we'll write our code. We'll work with the same message structure as before.

[Video description begins] *She shifts to the Composition.js tab. Ten code lines appear in lines 1-10.* [Video description ends]

Here is a message component that displays a message to screen. It's extremely simple. It's a stateless component. You can see that it has just a render function, and we apply some CSS styles to display this message.

[Video description begins] *She highlights code line 1. It reads: class Message extends React.Component {. She points at code line 3.* [Video description ends]

We apply the border CSS class, the plain-message class. And any additional CSS class specified in this.props.css class.

[Video description begins] *She highlights code line 5. It reads: <div className = {'border plain-message' + this.props.cssClass}>.* [Video description ends]

The message that is displayed to screen is also available as a part of the props this.props.message.

[Video description begins] *She highlights code line 6.* [Video description ends]

Again, a very straight-forward component. Next, I'll set up a success message component to display the same message but in a different style.

[Video description begins] *A set of code lines appear in lines 12-20. She highlights code line 12. It reads: class SuccessMessage extends React.Component {.* [Video description ends]

Success message will reuse the code available in the message component, but we'll use composition as a design strategy rather than inheritance. The message component will be composed within the success message component. Here is the render function of the success message component, all it does is render out the message component.

[Video description begins] *She highlights code lines 15-18.* [Video description ends]

Whatever props have been passed into success message we use the spread operator, that is the three dots, to pass in all of those props to the message component. In our example here, the props will include the message contents that we want displayed to screen.

[Video description begins] *She highlights code line 17. It reads: < Message { ... this.props} cssClass= 'success-message'></Message>.* [Video description ends]

In addition, we pass in another value for props, cssClass. In addition to whatever CSS styles the message component applies, the success-message cssClass should also be included for the success-message component. If you look at the message component up top, it accesses this.props.cssClass to add in any additional styles that have been passed into it.

[Video description begins] *She points at code line 5.* [Video description ends]

We'll now set up a Dialog class as we did earlier, which displays the message as well as the success message. So, message will display a plain message that is the text content SuccessMessage will display a message indicating success.

[Video description begins] *A set of code lines appear in lines 21-31.* [Video description ends]

[Video description begins] *She highlights code lines 26 and 27.* [Video description ends]

This is exactly the same as in the previous demo. All that's left to do is to use ReactDom.render, and render out this dialog component to our DOM.

[Video description begins] *The following code appears in line 33: ReactDOM.render(<Dialog />, document.getElementById('react-composition'));.* [Video description ends]

I'll now switch over to the Chrome browser and click through to Composition.html.

[Video description begins] *Two messages appear on the left. On the right, the Console tab is open.* [Video description ends]

And here are our two messages, a plain message and a message indicating success. If you look at the HTML structure of this page in the elements tab of Chrome Developer Tools, you'll see that the cssClasses have been correctly applied.

[Video description begins] *She shifts to the Element tab. It has an html script. She shifts back to the Composition.js tab.* [Video description ends]

Composition in React makes your code robust, maintainable, and easier to read. Here I've added in an additional Component called FailureMessage, which uses once again composition for code reuse. It has the message component nested within it.

[Video description begins] *A set of code lines appear in lines 21-28. The succeeding code lines shifts eight lines down.* [Video description ends]

Take a look at the render function of this failure message. We have the message component and we pass to this all of the props that the failure message receives from it’s higher level component. This dot props is passed into message using the spread operator. We also pass in the cssClass property error-message, indicating that that should be applied to the message that we display as well.

[Video description begins] *She highlights code line 25.* [Video description ends]

Now I'm going to scroll down to this dialog component and paste in this failure message component here as well.

[Video description begins] *She inserts a code line at line 37. The succeeding code lines shifts one line down.* [Video description ends]

The message it'll display will be a message indicating failure. Save the contents of this file and switch over to Chrome and hit refresh. And there they are, the three components that we set up in our code using composition rather than inheritance. Using composition rather than inheritance for react components allows you to reuse code in an elegant, robust and maintainable manner. The code is easier to read, easier to debug and thus easier to maintain.

**Using Composition to Customize Child Elements**

[Video description begins] *Topic title: Using Composition to Customize Child Elements. Your host for this session is Janani Ravi.* [Video description ends]

When you use composition as a design strategy to set up your React components, it's more than just about elegant, maintainable, and robust code. It's also about how you think about the component layout the component tree on your web page.

[Video description begins] *A Sublime text editor opens. The editor pane has two tabs: CompositionWithChildren.html and CompositionWithChildren.js. Currently, the CompositionWithChildren.html tab is open. It has some code lines.* [Video description ends]

It's quite possible that when you are designing your application, a component does not know what children will be rendered within it ahead of time, you have no idea what nested components will be within a component. It's still possible to enforce certain designs and styles within your nested component, if you use composition. Composition, as you know, is all about containment, nested structures that are contained within one another.

We can use composition to enforce design principles on the components that make up your application. Even when a component doesn't know its children up front, you can enforce certain design principles on those children and that's exactly what we'll do here in this demo. This is the HTML file for our application, we'll write our code in CompositionWithChildren.js. We'll see how we can use the .childrenproperty in props to define styles for the children of a component.

[Video description begins] *She shifts to the CompositionWithChildren.js tab. A set of code lines appear in lines 1-9.* [Video description ends]

Here I have a component called CustomBorderBackground. This is a component that allows me to customize the background color as well as the border for any children components nested within this component.

[Video description begins] *She highlights code line 1.* [Video description ends]

CustomBorderBackground extends the base React.Component class and within the render function, it simply applies some css styles to its children. This component here allows us to customize the border for any child component, we can specify a certain color for the border using this.props.color, the color property is passed into this component.

[Video description begins] *She highlights code line 4. It reads: <div className = {'bt bt-' + this.props.color + ' bg-' + this.props.color}>.* [Video description ends]

We'll also define the background for this component to be of a certain color and that color is available in this.props.color as well. This component does not have any additional elements of its own all it does is render whatever is present in this.props.children. If you remember, the .children property is automatically available as a part of this.props, allowing any component to access nested child components within it.

[Video description begins] *She highlights code line 5.* [Video description ends]

So whatever nested child component you've specified will have a custom background and a custom border. Let's see how we can use this CustomBorderBackground component to customize other components. I have another component here called Greeting, which extends from the React.Component base class and within its render function, I have some content that I want to display, I want to display an h1 header and some content in italics.

[Video description begins] *A set of code lines appear in lines 11-22. She highlights code line 11.* [Video description ends]

[Video description begins] *She highlights code lines 16 and 17. Line 16 is: <h1 className = "greeting-title">Good morning!</h1>. Line 17 is:<i className = "greeting-message">Have a great day..:-)</i>.* [Video description ends]

I want this content to have a customized background color as well as a customized border, so I surround this content using the CustomBorderBackground component.

[Video description begins] *Two code lines appear in lines 15 and 18.* [Video description ends]

The CustomBorderBackground component takes in just a single property in its props, color I've set to green and nested within it, we have our h1 header and an i tag for our content in italics.

[Video description begins] *She highlights code line 15.* [Video description ends]

[Video description begins] *She highlights code lines 16 and 17.* [Video description ends]

Both of these elements, the h1 header and the i tag are children of the CustomBorderBackground and these will be rendered within a customized background and a customized border. I'm now going to go ahead and invoke ReactDOM.render to render out this Greeting component, save this file and let's switch over to our Chrome browser and click through to CompositionWithChildren.html.

[Video description begins] *The following code appears in line 24: ReactDOM.render(<Greeting />, document.getElementById('react-composition-children'));.* [Video description ends]

[Video description begins] *On the left, a box titled Good morning! appears. On the right, the Console tab is currently open.* [Video description ends]

And here is a greeting that we specified, Good morning! and Have a great day. All within our customized background and customized border, both of which are in the green color. You can see how we used composition to apply this customization to the nested components, take a look at the HTML structure of the page in Chrome developer tools.

[Video description begins] *She shifts to the Elements tab. It has an html script.* [Video description ends]

You can see that the outer div comes from the custom border background component, you can see the CSS Styles applied to the outer div bt bt- green bg- green. And within this outer custom border background, we have the contents of the greeting component, the h1 header and our text in the italics tag. I'll now switch back to our Sublime Text editor. I'm still working in the class CompositionWithChildren.js and I'm now going to set up a more complex component. This component contains a form which you can use to submit an email address to the server.

[Video description begins] *A set of code lines appear in lines 24-33.* [Video description ends]

We'll see how we can customize this more complex component as well using the CustomBorderBackground component that we set up earlier. Once again, using composition, let's take a look at this EmailForm component which extends the base React.Component base class.

[Video description begins] *She highlights code line 24.* [Video description ends]

It hold some internal state, the current value for email that has been set on your form, it's a controlled component.

[Video description begins] *She highlights code lines 29-31. Line 29 is: this.state = {. Line 30 is: emailValue: ' ',. Line 31 is: };.* [Video description ends]

Any change or any edit that the user has made to the input text box will invoke this handleEmailChange event handler.

[Video description begins] *A set of code lines appear in lines 34-37. She points at code line 34. It reads: handleEmailChange = (e) => {.* [Video description ends]

Within this, we'll simply call this.setState, and set the latest state to be the email address specified in e.target.value. e.target will reference the input element where the user typed in the email address.

[Video description begins] *She highlights code line 36.* [Video description ends]

Within the render function of this component, we'll set up our form, you can see the form tag there.

[Video description begins] *A set of code lines appear in lines 39-53. She points at code lines 42 and 50.* [Video description ends]

Within the form, we have a div which contains the text email address and an input text box, an input element of type text.

[Video description begins] *She points at code lines 43 and 44. Line 43 is: <div className = "label">. Line 44 is: Email address:.* [Video description ends]

[Video description begins] *She points at code line 46. It reads: <input className = "input" type="text".* [Video description ends]

Since this is a controlled component, the value of this input text box is wired up to this.state.emailValue and onChange will invoke that this.handleEmailChange event handler.

[Video description begins] *She highlights code lines 47 and 48.* [Video description ends]

This entire form structure is composed within the CustomBorderBackground component and we have specified that the custom color for this CustomBorderBackground is red, color is equal to red is the props that we pass in. Observe how the nested elements within this CustomBorderBackground component is far more complex then just a simple header or an italic tag or a bold tag, it's an entire form.

[Video description begins] *She highlights code line 41.* [Video description ends]

But this works as well, no matter what children you have within the CustomBorderBackground component. Our form still needs to be wired up to a submit event handler, this is the handleSubmit function we'll simply print out to the console log the email address that was submitted available in this.state.emailValue.

[Video description begins] *A set of code lines appear in lines 39-45. The succeeding code lines shift seven lines down. She points at line 39. It reads: handleSubmit = (e) => {.* [Video description ends]

[Video description begins] *She highlights code line 40.* [Video description ends]

We then call this.setState to update the email value to an empty string, and e.preventDefault to prevent the default browser handling of this form.

[Video description begins] *She highlights code lines 42 and 44.* [Video description ends]

I'll now wire up the submit handler to the form.onSubmit property, onSubmit will invoke this.handleSubmit. I'll scroll down to the bottom here and comment out the greeting component and instead use ReactDOM.render to render out this email form component.

[Video description begins] *She edits code line 50.* [Video description ends]

[Video description begins] *The following code appears in line 24: ReactDOM.render(<EmailForm />, document.getElementById('react-composition-children'));.* [Video description ends]

I'll now switch over to the Chrome browser and hit Refresh, I want to see how my email form looks with the custom border and background that we have specified. You can see we have a little input box, the Submit button, all within a red background and a red border. If you look at the highlighted div in the HTML structure of your page off to your right, you can see that the custom border background div has the class bt, bt-red, and bg-red applied. You can of course take this form for a test drive to see whether it works as you expect, here is the email address that I want to submit to the server. Hit Submit and the email address bob@skillsoft.com has been submitted successfully.

[Video description begins] *She shifts to the Console tab.* [Video description ends]

**Using Composition for Specialization**

[Video description begins] *Topic title: Using Composition for Specialization. Your host for this session is Janani Ravi.* [Video description ends]

If you come from an object-oriented programming background and you work with the programming language such as Java, you often think of inheritance for specialization. The base class is a more generic class, and derived classes are more specific, such as you may have a base shape class and derived classes will be triangle, rectangle, and so on.

[Video description begins] *A Sublime text editor opens. The editor pane has two tabs: CompositionForSpecialization.html and CompositionForSpecialization.js. Currently, the CompositionForSpecialization.html tab is open. It has some code lines.* [Video description ends]

When you're working in React though, you'll always prefer composition over inheritance. And the same specialization can be achieved using composition in React. When you're working in React, you'll use composition for specialization as well. You'll have a more generic component that has the ability to take in a number of values as input via props. You'll then customize this generic component using props to use for specialized cases.

So you'll have a more specific component color mode, generic component by customizing it. And this is exactly what we'll understand and implement here in this demo. CompositionForSpecialization.html is the name of this file, we'll write all of our code in CompositionForSpecialization.js. Let's head over to the JS file and get started coding.

[Video description begins] *She shifts to CompositionForSpecialization.js tab. A set of code lines appear in lines 1-14.* [Video description ends]

The first component that we'll set up here will be a CustomDialog that will serve as our most generic component.

[Video description begins] *She highlights code line 1.* [Video description ends]

This is the component that can be customized using props. The class CustomDialog extends the base React.Component class and within its render function, I've assigned a variable color to be green.

[Video description begins] *She highlights code line 4.* [Video description ends]

The elements of this CustomDialog are an outer div that can be customized based on the color property that we've set. It will have a custom border color and a custom background color based on this color property.

[Video description begins] *She highlights code line 7. It reads: <div className = {'bt bt-' + color + ' bg-' + color}>.* [Video description ends]

Within this, there is a header and a div with content. The h3 header of the dialog will display the text that is contained in this.props.title, and the content will be available in this.props.message.

[Video description begins] *She highlights code line 8.* [Video description ends]

That's what will be displayed.

[Video description begins] *She highlights code line 9. It reads: <div className = "dialog-message">{this.props.message}</div>.* [Video description ends]

This CustomDialog component allows us to customize the title, the message displayed, as well as the color of the border and background. We'll now setup yet another component which is a specialized component. This is the OKDialog that extends the base React.Component class.

[Video description begins] *A set of code lines appear in lines 16-25. She points at line 16.* [Video description ends]

But nested within it, it invokes the CustomDialog component, and customizes it by passing in the right values for props. Within the render function of the OKDialog, we have the CustomDialog, the dialogtype is ok. The title that we want to show is success, and the message to display is Your operation was completed successfully.

[Video description begins] *She points at code lines 20 and 21.* [Video description ends]

The dialogtype attribute here is what will determine the background color and the border color of the component. The default is green, and we'll leave it at that for now. We'll first take a look at what this component looks like on the browser before we make additional changes. Invoke ReactDOM.render, and let's render this OKDialog into the DOM of our HTML page.

[Video description begins] *Two code lines appear in lines 28 and 29. Line 28 is: ReactDOM.render(<OKDialog />, . Line 29 is: document.getElementById('react-composition-specialization'));.* [Video description ends]

Make sure you save this file and let's switch over to the Chrome browser and click through to CompositionForSpecialization.html.

[Video description begins] *A dialog box titled Success appears on the left. On the right, the Console tab is open.* [Video description ends]

Here is our specialized OKDialog. We created this using composition and composition allowed us to customize a more generic component, the CustomDialog that we had created. Now if you'll look at the element structure of this HTML page, you will be able to view the customization that we specified to the CustomDialog.

[Video description begins] *On the right, she shifts to the Elements tab. It has an html script.* [Video description ends]

We had initialized the color variable to green, bt-green and bg-green is the CSS class that we have applied to this outer div. The dialog-title displays Success, and the message says, Your operation was completed successfully. Let's switch back to our Sublime text editor and work to improve our CustomDialog.

[Video description begins] *She shifts to the CompositionForSpecialization.js tab. A set of code lines appear in lines 7-11. The succeeding code lines shifts five lines down.* [Video description ends]

I'm going to have this component read the dialogtype property available in this.props. If this.props.dialogtype is warning, the color that we'll use to customize this component will be orange.

[Video description begins] *She highlights code lines 7 and 8.* [Video description ends]

Or if this.props.dialogtype is equal to error, the color that we'll use will be the red color.

[Video description begins] *She highlights code lines 9 and 10.* [Video description ends]

In addition to the OKDialog, we now have the ability to support two more specialized dialogs, the warning dialog and the error dialog. Let's add those components in as well. I've scrolled down to the bottom here and here is our WarningDialog component.

[Video description begins] *A set of code lines appear in lines 35-45. The succeeding code lines shifts eleven lines down.* [Video description ends]

The WarningDialog, once again, uses composition for specialization. It's a special kind of dialog, but we invoke the CustomDialog component within it and configure this CustomDialog by passing the right value in the props. dialogtype = 'warning' title says warning and the message says, 'Your operation was completed, but there were a few errors.'

[Video description begins] *She points at code line 35. It reads: class WarningDialog extends React.Component {.* [Video description ends]

[Video description begins] *She highlights code lines 39-41.* [Video description ends]

I'm going to get rid of this OKDialog and now use ReactDOM.render and render out this WarningDialog to my DOM. I want to see what it looks like.

[Video description begins] *She edits code line 47. It now reads: ReactDOM.render(<WarningDialog />,.* [Video description ends]

Having saved my changes, I'm going to switch back to Chrome once again and hit refresh. This was a dialog that was shown previously, the OKDialog.

[Video description begins] *A dialog box titled Success is present on the left. On the right, the Elements tab is open.* [Video description ends]

But now on refresh, we get the WarningDialog. The WarningDialog is displayed using an orange background and an orange border. You can take a look at the HTML structure and see the classes that are applied to this div. You can see that the div class for our CustomDialog uses the CSS styles bt-orange and bg-orange to customize this. I'm going to switch back to the Sublime text editor here and create one more specialized component using composition.

[Video description begins] *A set of code lines appear in lines 47-57. The succeeding code lines shifts eleven lines down.* [Video description ends]

This is the ErrorDialog class, which uses the CustomDialog class and customizes it to display an error.

[Video description begins] *She points at code line 47. It reads: class ErrorDialog extends React.Component {.* [Video description ends]

Within the render function of this ErrorDialog, we have the CustomDialog component, we specified dialogtype is equal to error. The title says Error!, and the message says, 'Your operation could not be completed.' Thus, we have a specialized dialog which is created using composition, using a more generic component.

[Video description begins] *She highlights code lines 51-53.* [Video description ends]

I'm now going to change my ReactDOM.render function to render out this ErrorDialog.

[Video description begins] *She edits code line 59. It now reads: ReactDOM.render(<ErrorDialog />,.* [Video description ends]

Make sure you save this file. And let's switch over and refresh our Chrome browser. And here is our ErrorDialog. The ErrorDialog is a specialized component with a red background and a red border. If you take a look at the HTML structure for this dialog, you can see that the CustomDialog has the CSS class bt-red and bg-red applied in order to generate the custom border and background.

**Global Properties without Context**

[Video description begins] *Topic title: Global Properties without Context. Your host for this session is Janani Ravi.* [Video description ends]

When you follow best practices in React, you will have data flow from higher level components to lower level components via props. When you're working in React, state should always be stored in the highest possible component in the component tree. And prop should be passed down to individual components to configure how they're displayed.

[Video description begins] *A Sublime text editor opens. The editor pane has two tabs: NoContext.html and NoContext.js. Currently, the NoContext.html tab is open. It has some code lines.* [Video description ends]

Sometimes though, you might have a value or a property that is specified across your app. There might be a number of nested components within your app that require that property to render themselves, to update their state, or change their behavior. Now, passing down such a property through the nested component tree can be rather hard. React allows you to use something called the context to allow props to flow through different components in your component tree without having to pass them manually.

But before we work with context, let's see a simple application where we have no context, which means we have to manually pass down properties that are applicable to many components. NoContext.html is the name of our HTML file. NoContext.js is the JavaScript file that we'll edit. The app that we are going to build here will have two different themes, light and dark. And every component that you write within this app will have different displays for the light theme and the dark theme.

[Video description begins] *She shifts to the NoContext.js tab. Ten code lines appear in lines 1-10.* [Video description ends]

Here is the first component that we have within our application. This is a ThemedBorderBackground.

[Video description begins] *She points at code line 1. It reads: class ThemedBorderBackground extends React.Component{.* [Video description ends]

It specifies a customized border based on the theme that you've specified and a customized background as well for that same theme. This is a stateless component with just a render function. It displays a div with a customized colored background and border. The color is determined based on this.props.theme. And the background is also based on this.props.theme. And the theme is passed in as a part of props.

[Video description begins] *She highlights code line 5. It reads:<div className = {'bt bt-' + this.props.theme + ' bg-' + this.props.theme}>.* [Video description ends]

This component simply renders whatever children have been specified for this using this.props.children using a customized border and a customized background.

[Video description begins] *She points at code line 6.* [Video description ends]

Let's take a look at another component that accepts a theme.

[Video description begins] *A set of code lines appear in lines 12-22.* [Video description ends]

This component is a ThemedButton. It extends the base React.Component.

[Video description begins] *She highlights code line 12.* [Video description ends]

This is a stateless component, which just renders a style button to screen. The style of the button is based on the theme specified. You can see that the CSS class applied uses this.props.theme.

[Video description begins] *She points at code line 16. It reads: <button className = {'button button-' + this.props.theme}>.* [Video description ends]

The text content displayed in the button is available in this.props.label. Also a property passed in from a higher level component.

[Video description begins] *She points at code line 17.* [Video description ends]

So far we have two components in this application. But in the real world, you might have many such components. Each needs a theme property to determine how they will be rendered. I'll now use ReactDOM.render to render both of these components to screen.

[Video description begins] *A set of code lines appear in lines 24-34.* [Video description ends]

We have the ThemedBorderBackground, and the theme that I have specified here is the theme light. The only text content that it'll display will be the word Hello!

[Video description begins] *She points at code lines 26 and 27.* [Video description ends]

Just below that component, I have the ThemedButton, and the theme that I've specified here is light once again. And the button will display the label Click me!

[Video description begins] *She points at code line 30.* [Video description ends]

Save the contents of this file. We'll head over to the Chrome browser where we have our HTTP server running. Click through to NoContext.html.

[Video description begins] *On the left, a Hello! message appears. A Click me! button is present below it. On the right, the Console tab is currently open.* [Video description ends]

And here are two components displayed using the light theme. We have the Hello with a yellowish background. And a Click me button with the yellowish background as well. If you take a look at the HTML structure of the components that we've displayed.

[Video description begins] *In the right pane, she shifts to the Elements tab.* [Video description ends]

You can see that the CSS classes for the two components include the CSS styles for the light theme bt-light, bg-light are the border and the background themes. And for the button we have the button-light CSS class applied. We've set up our CSS such that when the theme is the light theme, when you hover over the button, it'll become a slightly darker color.

[Video description begins] *She shifts back to the NoContext.js tab.* [Video description ends]

Having the theme passed in as a part of the props for each component has been manageable so far. Because we only have two components. But let's add one more component. Let's say you have a dialog that can also be themed and accepts a theme as a part of its props.

[Video description begins] *A set of code lines appears in lines 24-38. The succeeding code lines shifts fifteen lines down.* [Video description ends]

This ThemeDialog here is a stateless component. Once again, you can see that the outer div of this dialog applies a CSS class based on this.props.theme.

[Video description begins] *She highlights code line 27. It reads: <div className = {'dialog dialog-' + this.props.theme}>.* [Video description ends]

The dialog title will be displayed using an h3 header. And the content for this title will be in this.props.title.

[Video description begins] *She highlights code line 29.* [Video description ends]

The rest of the content of the dialog will be within a div, and the content will be available in this.props.message.

[Video description begins] *She highlights code line 30. It reads: <div className = "dialog-message">{this.props.message}</div>.* [Video description ends]

Now, this ThemeDialog component has nested within it the ThemedButton component. Remember, the ThemedButton component also needs the theme property to be passed in. So we need to pass the theme into the ThemedButton as well. this.props.theme is a part of the props for the ThemedButton.

[Video description begins] *She points at code line 33. It reads: <ThemedButton theme={this.props.theme} label = 'OK'>.* [Video description ends]

It should be pretty obvious to you by now that as your React application becomes more complex and includes more components, passing the theme down in the form of props can get very messy. You'll have to propagate it to every component that is themed. I'm now going to render this ThemedDialog here using ReactDOM.render.

[Video description begins] *Three code lines appear in lines 49-51.* [Video description ends]

This ThemedDialog will also be displayed using theme='light'. The title will be Dialog title here. And the message will simply say, Dialog message here.

[Video description begins] *She points at code lines 49 and 50.* [Video description ends]

Save this file and let's go back to our Chrome browser. Hit the refresh button. And here is our updated page with the themed dialog at the very bottom.

[Video description begins] *She shifts back to the browser. In the left pane, a dialog box appears.* [Video description ends]

The themed dialog has the dialog title and the dialog message in a yellowish background. And it also has a themed button within it which says OK. You can look at the HTML structure of the page. And confirm for yourself that the theme applied here, all of the CSS classes have the suffix -light. Now, the same light theme is what we have applied for each of the components that we have rendered. And this is typically the case. All your components within your React application will follow the same theme. I'm going to go back to my Sublime Text Editor. And I'm going to change the theme that I've used to display each component.

[Video description begins] *She shifts back to the NoContext.js tab. She points at code lines 42, 46, and 49.* [Video description ends]

Now I'll have to go in and change the props passed into each component and switch the theme over to the dark theme. It's kind of tedious having to make this change for each component. And it's unmanageable in a large application. Save this file, and hit refresh once again on Chrome browser. And here are all of the components displayed using the dark theme. If you take a look at the HTML structure of the page, you'll find that all of the HTML elements remain the same. It's only the CSS styles that have been applied to each element that are different. All of the CSS styles end with the suffix -dark, indicating the dark theme.

**Using Context to Specify Global Properties**

[Video description begins] *Topic title: Using Context to Specify Global Properties. Your host for this session is Janani Ravi.* [Video description ends]

If you have values that need to be propagated to different components at different levels of the component tree, and it's kind of messy doing it using props. Well, React has a way around for you, you can use Context.

[Video description begins] *A Sublime text editor opens. The editor pane has two tabs: Context.html and Context.js. Currently, the Context.html tab is open. It has some code lines.* [Video description ends]

The use of Context within React allows you to pass data through the component tree without manually having to pass this data using props at each level. Passing data through props can be cumbersome for certain kinds of values that are required by all components or many components within your application, such as the locale preference or themes.

The use of Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree. In this demo, that we'll write in Context.html and Context.js, we'll build the same components that we saw in the previous demo. But this time we'll theme those components using context rather than passing props manually. Instead of passing the theme for our components as a part of the props,

[Video description begins] *She shifts to the context.js tab. Thirteen code lines appear in lines 1-13.* [Video description ends]

I'm now going to specify a context for this theme. I've initialized a new context here in the const ThemeContext using React.createContext. And the default specification for the theme will be the light theme, which is what we've passed into React.createContext.

[Video description begins] *She highlights code line 1.* [Video description ends]

Next you can see the ThemedBorderBackground component. This is the same component that we saw in an earlier demo, it extends the base React.Component class.

[Video description begins] *She highlights code line 3.* [Video description ends]

Within the render function for this component, things are a little bit different.

[Video description begins] *She points at code line 4.* [Video description ends]

When we display this component using a custom border and custom background based on theme, we don't get the theme using props, we access the current context using this.context.

[Video description begins] *She highlights code lines 7-9. Line 7 is: <div className = {'bt bt-' + this.context + ' bg-' + this.context}>.* [Video description ends]

You can see that the CSS class uses this.context to get the theme that needs to be applied. The ThemedBorderBackground component will simply render all of its children within the customized background and with the customized border. In order to be able to access the context from within a React component, that is access the context using this.context, you need to set the context type property for this component.

ThemedBorderBackground.contextType is the ThemeContext that we had set up earlier. The light theme is the default theme that we have chosen for our application. I'm now going to scroll down and set up the ThemedButton component that we saw in an earlier demo. And this in turn will use the context as well, to get the theme that it needs to be displayed in. The ThemedButton's render function access the current context using this.context to specify the CSS style that needs to be applied. We'll specify the default context for the theme button using ThemedButton.contextType which we set to the ThemeContext that we had created earlier.

[Video description begins] *Thirteen code lines appear in lines 16-28. She highlights code lines 20-22. Line 20 is:<button className = {'button button-' + this.context}>. Line 21 is:{this.props.label}. Line 22 is: </button>.* [Video description ends]

I'm going to scroll down a little further and set up the ThemedDialog component that we have seen in an earlier demo. And this ThemedDialog will also get its theme using the current context.

[Video description begins] *Seventeen code lines appear in lines 30-46.* [Video description ends]

You can see that the outer div of this ThemedDialog within the render function accesses this.context to get the CSS style that needs to be applied.

[Video description begins] *She points at code lines 31 and 33. Line 31 is: render() {. Line 33 is: <div className = {'dialog dialog-' + this.context}>.* [Video description ends]

The rest of this render function remains the same. A notable difference is in the ThemedButton. We have a ThemedButton component within the ThemeDialog, but we don't need to pass the theme in explicitly as a part of the props.

[Video description begins] *She highlights code lines 39 and 40. Line 39 is: <ThemedButton label='OK'>. Line 40 is: </ThemedButton>.* [Video description ends]

The default context that the ThemedDialog uses needs to be specified using the contextType property. We set ThemedDialog.contextType to the ThemeContext that we had initialized earlier. All of the components that we've set up here in this file use the current theme specified in the ThemeContext as the default. That is the theme in which they will be rendered by default. And the default rendering is what we'll show first using ReactDOM.render.

[Video description begins] *Twelve code lines appear in lines 48-59. She points at code line 48.* [Video description ends]

We'll display a ThemedBorderBackground with Hello!, a ThemedButton with the label Click me!, and the ThemedDialog with the title and a message.

[Video description begins] *She highlights code lines 50-57. Line 50 is: <ThemedBorderBackground>. Line 51 is: Hello!. Line 52 is:</ThemedBorderBackground>. Line 53 is: <ThemedButton label='Click me !'>. Line 54 is: </ThemedButton>. Line 56 is: <ThemedDialog title='Dialog title here' message='Dialog message here'>. Line 57 is: </ThemedDialog>.* [Video description ends]

We don't have to specify the theme explicitly for any of these components. They will pick up by default the theme that we have specified in the ThemeContext. I'll now head over to the Chrome browser that I have open and click on Context.html to see what my UI looks like.

[Video description begins] *A webpage titled Index of /Project/React/Context/Context/ opens in a browser. There are links for two files: Context.html and Context.js. A Developer tools pane is open on the right. The Console tab is currently open.* [Video description ends]

[Video description begins] *In the left pane, two fields appear: Hello! and Dialog title here. Under Hello! there is a Click me! button. Under Dialog title here, an OK button is present.* [Video description ends]

Here I have the three components that we rendered, all of them rendered using the light theme, which they got from the default context that we had specified.

[Video description begins] *In the right pane, she shifts to the Elements tab. It has some code lines.* [Video description ends]

If you look at the HTML structure of this page, you'll find that the CSS styles apply to all of these components end with the suffix -light, indicating the light theme, picked up from the default theme. Let's head back to our Sublime Text Editor here and see how we can use a ThemeContext.Provider to change the theme that we want our components to be displayed in. A context provider in React allows you to specify a value for some context that can then be consumed by the child components.

[Video description begins] *She shifts back to the context.js tab. The following code appears in line 50: <ThemedContext.Provider value="dark">. The succeeding code lines shift one line down.* [Video description ends]

I have specified the ThemeContext.Provider here. And the children of this ThemeContext.Provider include the ThemedBorderBackground, the ThemedButton, as well as the ThemedDialog.

[Video description begins] *She points at code lines 51, 54, and 57.* [Video description ends]

The value specified by this ThemeContext.Provider is value= dark. So the closest context for all of these components is the dark theme. So all of these components will no longer use the default context, instead they will use the dark theme, which is the closest context. Hit refresh on this page, and you'll see the change immediately.

[Video description begins] *She shifts back to the browser.* [Video description ends]

All of the components are now rendered using the dark theme, because that was the theme value specified by the ThemeContext.Provider surrounding these components. Let's go back to our Sublime Text Editor and make another change.

[Video description begins] *She shifts back to the context.js tab.* [Video description ends]

I'm now going to move this ThemeContext.Provider to only surround the ThemedDialog. With this change, the ThemedDialog will get its ThemeContext using the ThemeContext.Provider.

[Video description begins] *She inserts a code line at line 55. It reads: <ThemedContext.Provider value="dark">. The succeeding code lines shifts one line down. The following code appears in line 58: </ThemedContext.Provider>.* [Video description ends]

But the ThemedBorderBackground and the ThemedButton will get its context from the default light theme that we had set up earlier. Let's check this using Chrome. I'm going to hit Refresh on this page, and you'll see that the first two components here are displayed using the light theme, the dialog is displayed using the dark theme.

[Video description begins] *She shifts back to the browser. In the left pane, Hello! and Click me! appear in light background.* [Video description ends]

Our ThemedDialog gets its context from the surrounding ThemeContext.Provider with value="dark". The remaining components, the ThemedBorderBackground, and the ThemedButton, get their context from the default ThemeContext that we initialized, which was set to the light theme. Let's head back to our Sublime Text Editor and make another change. I'm going to move this ThemeContext.Provider to surround just the ThemedBorderBackground component.

[Video description begins] *She inserts a code line at line 50. The succeeding lines shift one line down.* [Video description ends]

So the ThemeContext.Provider will not surround all of our UI, but only will surround the ThemedBorderBackground.

[Video description begins] *She inserts a code line at line 54. The succeeding code lines shift one line down.* [Video description ends]

TheThemedBorderBackground component should pick up its theme from the Context.Provider. The remaining component should use the default ThemeContext. Let's hit Refresh on this page and see whether this is true, and yes, indeed it is.

[Video description begins] *She shifts back to the browser.* [Video description ends]

The first component on screen, the ThemedBorderBackground, gets its theme value from the ThemeContext.Provider, which has a value set to dark.

[Video description begins] *On the left, Hello! appears in dark background. Click me!, OK, and dialog field appear in light background.* [Video description ends]

The remaining components, the ThemedButton and the ThemedDialog, get its theme value from the default that we had specified, the ThemeContext, initialized to light. Context in React is great and has many uses. It lets you broadcast data to all components irrespective of where they are in the component tree. However, in the real world, make sure you use Context sparingly. Context serves as kind of a global reference for multiple components, and this can make components harder to reuse. So keep that in mind when you use Context in your applications.

**Course Summary**

[Video description begins] *Topic title: Course Summary.* [Video description ends]

In this course, we covered a range of Advanced Features in React. We began by understanding the error that occurs when list elements are rendered without keys. We saw how we could specify unique keys for list elements. And we saw the correct usage of keys in components. We then moved on to the important topic of conditional rendering. This included rendering components using conditional evaluation, performing conditional rendering using variables and evaluating conditions using inline, logical and ternary operators.

We then moved on to the ideas of code re-use, as well as lifting state to higher level components. We studied code re-use using inheritance, as well as composition. And the use of global properties, both with and without context. We are now ready to move on to a simple React application that we'll prototype in a development environment and productionize using create react app tools.