

1. **switch**开关用法, 使用**state**更新状态和文字
2. **condition expression (statement ? Y : N)**
3. 使用**switch**更新**style**, **style override**
4. **extract** 成员变量 **let {member1, member2} = object;**
5. **textInput**用法

06. User Input, Conditional Display

We've seen how to respond to user taps, but how do we collect data from the user and store it for use in our program? In this section we'll look at how to work with React Native components that support user input.

Let's start with the simplest input component: a toggle switch.



This is a component that can only have two possible states: `true` and `false`. You can create a toggle switch and set its value in React Native like this:

```
<Switch value={true}/>
```

Go ahead and navigate to your `week6Switch` project, change `App` to a class, and add a `Switch` to the `render()` method that is set to `true`. Remember to import `Switch` from `react-native`!

```
export default class App extends React.Component {  
  render() {  
    return (  
      <View style={style.container}>  
        <Switch value={true}/>  
      </View>  
    );  
  }  
}
```

Change the value to `false` in your code and save. What happens?

```
<Switch value={false}/>
```

In either case, what happens when you try to flip the switch? What do you think is going on?

What's going on is that right now there is nothing in the *code* that will *change* the value. In order to get the switch to stay in a new state, we need to capture the user input event and change the value. We can do this by handling the input event and changing the component's `state`, which is something we're familiar with by now.

First, we need to know how to capture the user input event. We do that by assigning a handler function to the `Switch`'s `onValueChange` property.

```
<Switch
  onValueChange={this.handleValueChange}
  value={true}/>
```

We can now set the `value` to a property of `this.state`, so that we can change how the `Switch` is displayed based on it's current (dynamic) value

```
<Switch
  onValueChange={this.handleValueChange}
  value={this.state.switchValue}/>
```

And now, we can initialize `this.state` in a constructor...

```
constructor() {
  super();
  this.state = {
    switchValue: false,
  }
}
```

And implement `handleValueChange` to update the state when the user taps the switch. Note that `handleValueChange` will take an argument, which is the current value *after* the user taps the button.

```
handleValueChange = (value) => {
  this.setState({switchValue: value});
}
```

```
}
```

Try it out!

Your entire App.js should look like this:

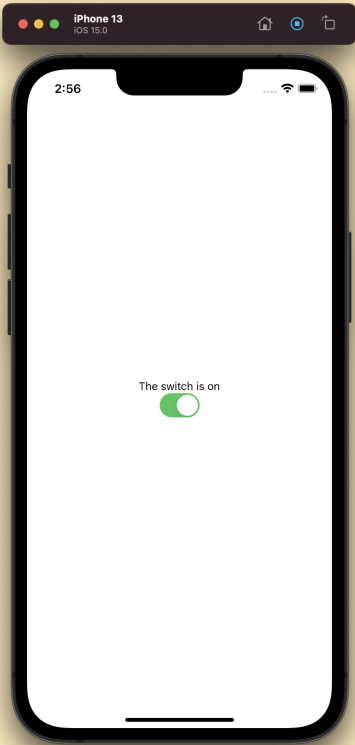
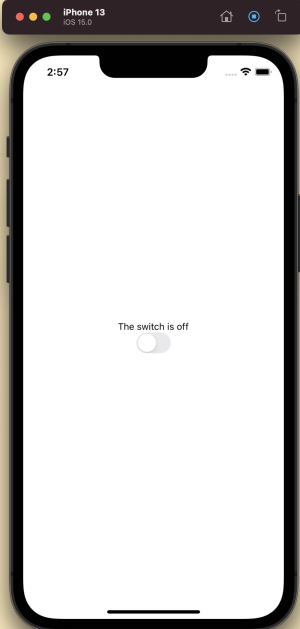
```
import React from 'react';
import { StyleSheet, Text, View, Switch } from 'react-native';

export default class App extends React.Component {
  constructor() {
    super();
    this.state = {
      switchValue: false
    }
  }
  handleValueChange = (value) => {
    this.setState({
      switchValue: value
    });
  }
  render() {
    return (
      <View style={style.container}>
        <Switch
          onChange={this.handleValueChange}
          value={this.state.switchValue}/>
      </View>
    );
  }
}

const style = StyleSheet.create({
```

```
container: {  
  flex: 1,  
  backgroundColor: '#fff',  
  alignItems: 'center',  
  justifyContent: 'center',  
}  
});
```

Yay! The switch actually works like a switch should. Other than, not very exciting.

Now You Try #1		
<ul style="list-style-type: none">• Add a text field that changes to reflect the current state of the switch		
Paste your App classes into slido .	How to fool Slido: In your <code>render()</code> function, <i>disrupt the parser</i> by adding	

an invalid character so that it doesn't see

```
<Tag attribute={something}>
```

but instead sees

```
<Tag attribute/= {something}>
```

Annoying, but something it seems we'll have to live with.

Let's make things slightly more interesting by having our switch toggle between "light mode" and "dark mode."



To make this work, we'll have to make the switch value control the text of the label ("Dark Mode: Off/On"--like you just did) as well as the styling of both the background and the label. There are several ways to do this, and we'll look at a few, but for our first try, let's add properties to our `state` property that will represent the label text and the overall `style` object to use for the entire display.

```
export default class DarkModeApp extends React.Component {  
  constructor() {  
    super();  
    this.state = {  
      switchValue: false,  
      switchStatus: 'Off',
```

```

        style: lightStyle,
    }
}
onValueChange = (value) => {
    // TBD
}
render() {
    return (
        <View style={this.state.style.container}>
            <Text style={this.state.style.text}>
                Dark Mode: {this.state.switchStatus}
            </Text>
            <Switch
                onValueChange={this.onValueChange}
                value={this.state.switchValue}/>
        </View>
    );
}
}

const lightStyle = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: 'white',
        alignItems: 'center',
        justifyContent: 'center',
    },
    text: {
        color: 'black'
    }
});

```



```
const darkStyle = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'black',
    alignItems: 'center',
    justifyContent: 'center'
  },
  text: {
    color: 'white'
  }
});
```

Note that the `style` properties for both the top-level `<View>` and the `<Text>` have been made variable by assigning them to `this.state.style.XXX`. This means that if we reassign `this.state.style` to either `lightStyle` or `darkStyle` when the switch changes state, the components will adopt the appropriate style. Note also that `this.state.switchStatus` will need to be updated with the appropriate status string ("Off" or "On"), and that this will cause the contents of the `<Text>` component to be updated as well. We can take care of all of this in the `onValueChange` function:

```
onValueChange = (value) => {
  let status = 'Off'; // default to Off
  let theStyle = lightStyle;
  if (value) {          // change to On if value==true
    status = 'On';
    theStyle = darkStyle;
  }
  this.setState({      // update state with value, status, and style
    switchValue: value,
    switchStatus: status,
    style: theStyle
  });
};
```

```
});  
}
```

Conditional Display

The Switch example above works just fine, and illustrates that you can use the `state` property of your `Components` to control a variety of aspects of how your UI will display. We've seen how it can control text that is displayed and styles that are applied, but now we'll look how to use `conditional display` and `state` to determine what to display and how.

The basic idea behind conditional display is that you write code *inside the render function* that determines what will be displayed. This sounds straightforward, and it is, but one aspect of JSX makes this a bit tricky. For example, you might think that it would be possible to do something like this:

```
return (  
  <View style={oneStyle.container}>  
    <Text>  
      Dark Mode: {  
        if (this.state.switchValue) {  
          "On";  
        } else {  
          "Off";  
        }  
      }  
    </Text>  
    <Switch  
      onChange={this.onValueChange}  
      value={this.state.switchValue}/>  
  </View>  
);
```

jsx无法在render中添加逻辑

And this would make logical sense—basically you’re saying that if `switchValue` is `true`, display one thing but if it’s `false` display another. The reason this won’t work, however, is that

```
if (this.state.switchValue) {  
  "On";  
} else {  
  "Off";  
}
```

is a valid (but slightly weird) JavaScript *statement*, but it isn’t an *expression*.

You may recall that anything that appears within `{ }` inside of a JSX block needs to resolve to a JavaScript *expression*. An expression in JavaScript is a snippet of code that resolves to a *value*. Expressions are distinct from *statements* in that the latter actually do things. Put differently, *statements* are instructions (“iterate over this array,” “assign this value to this variable), whereas *expressions* are just values (“1”, “hello world,” “false”).

Here are some examples:

Expressions

```
5; // resolves to 5  
5 + 6; // resolves to 11  
myFunction(); // resolves to the return value of myFunction()  
((x > 5) && (x < 10)); // resolves to true or false, depending on x
```

Statements

```
x = 5; // assigns the value 5 to x  
x = 5 + 6; // assigns 11 to x  
result[3] = myFunction(); // assigns myFunction()'s return value to result[3]  
if ((x > 5) && (x < 10)) { console.log("Just right!"); } // prints, maybe
```

So how would we change our first attempt to correctly use a JavaScript expression?

One way would be to assign a variable outside of JSX and then reference it from within the JSX. A variable name by itself is a valid expression—it resolves to a value. This would work:

```
render() {  
  let modeText = "Off";  
  if (this.state.switchValue) {  
    let modeText = "On";  
  }  
  return (  
    <View style={oneStyle.container}>  
      <Text>  
        Dark Mode: {modeText}  
      </Text>  
      <Switch  
        onChange={this.onChange}<br>  
        value={this.state.switchValue}/>  
    </View>  
  );  
}
```

在return之前定义变量

Note that we have included a JavaScript *statement* (`if ... else`) *outside* the JSX block and only included a valid expression inside of it.

You might note that this approach isn't really so different from what we did in the previous example, when we used a second `state` property (`switchStatus`) to represent the text that would be displayed. One advantage of using conditional rendering as we've done here is that it keeps all of your display code (i.e., "View") separate from your logic (i.e., "Controller") as well as your application state (i.e., "Model"). However, this is still a bit clunky in that the code that determines what will

be displayed is disconnected from the display code itself. To tighten things up a bit, we can make use of JavaScript's conditional operator.

The Conditional Operator

The conditional operator allows you to write an *expression* whose value is conditional. This is different from a conditional *statement* (`if...else`), which allows you to select different statements to execute based on a condition. The conditional operator works like this:

```
(condition) ? (value if true) : (value if false)
```

For example, consider the following:

```
let x = 5;
let xSize = (x < 10) ? "Small" : "Big";
console.log(xSize); // will print "Small"
```

The snippet `(x < 10) ? "Small" : "Big";` is an *expression*, which you will recall is some code that resolves to a value. The value of this expression depends on the truth or falsity of the conditional expression `x < 10`, and will resolve to either "Small" or "Big" accordingly.

We can use conditional operators to perform conditional rendering, like so:

```
render() {
  return (
    <View style={styles.container}>
      <Text>
        Dark Mode: {this.state.switchValue ? "On" : "Off"}
      </Text>
      <Switch
        onChange={this.onValueChange}
        value={this.state.switchValue}/>
    </View>
  );
}
```

```
);  
}
```

And we can even apply this strategy to choose the appropriate style, based on the value of the Dark Mode switch. Assume we have this `styles` object:

```
const oneStyle = StyleSheet.create({  
  container: {  
    flex: 1,  
    alignItems: 'center',  
    justifyContent: 'center',  
    backgroundColor: 'white',  
  },  
  bgDark: {  
    backgroundColor: 'black'  
  },  
  textLight: {  
    color: 'black'  
  },  
  textDark: {  
    color: 'white'  
  }  
});
```

We can rewrite `render()` to switch to/from dark mode like so:

```
render() {  
  return (  
    <View style={[oneStyle.container,  
      this.state.switchValue ?  
        oneStyle.bgDark :  
        {} ]}>
```

```

    <Text style={this.state.switchValue ?
      oneStyle.textDark :
      oneStyle.textLight}>
      Dark Mode:
      {this.state.switchValue ?
        "On" :
        "Off"
      }
    </Text>
    <Switch
      onChange={this.onValueChange}
      value={this.state.switchValue}/>
  </View>
);
}
}

```

Let's look a bit more closely at the definition of the outer `<View>` component, which looks like this:

```

<View style={[oneStyle.container,
  this.state.switchValue ?
    oneStyle.bgDark :
    {} ]}>

```

There's something else going on here that's new, which is that we're defining a *list of styles* instead of just a single style object. This is legit, and if there are styles later in the list that contain the same properties as styles earlier in the list, the later style will override the earlier one. In the example we have here, the first item in the style list is always `oneStyle.container`. The second item in the list will be determined by the expression

`this.state.switchValue ? oneStyle.bgDark : {}`, thus if `switchValue` is `true`, the style list will be `[oneStyle.container, oneStyle.bgDark]`. In this case, the `backgroundColor` property in `bgDark` (which is `'black'`) will override the `backgroundColor` in `container` (which is `'white'`). If `switchValue` is `false`, on the other hand, the `View`'s style list will be `[oneStyle.container, {}]`, with an empty object as the second item. In this case, the empty object has no properties, so will override nothing, and `backgroundColor: 'white'` will determine the `View`'s color.

Now You Try #2

Fill in the missing code (`/* what goes here? */`) below with a conditional expression that will cause the words “Dark Mode: On/Off” to appear correctly in both dark and light modes. Refer to the `oneStyle` object defined above for the style names to use.

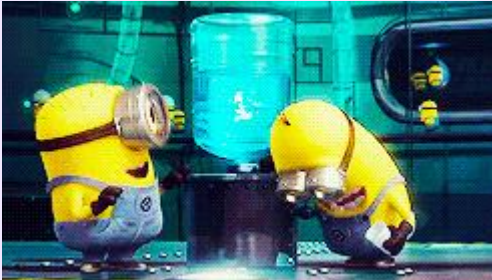
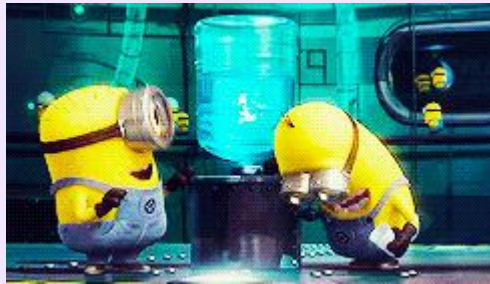
Enter your conditional expression into [slido](#).

```
render() {
  return (
    <View style={[oneStyle.container,
      this.state.switchValue ?
        oneStyle.bgDark :
        {} ]}>
      <Text style={/* what goes here? */}>
        Dark Mode:
        {this.state.switchValue ?
          "On" :
          "Off"
        }
      </Text>
      <Switch
        onChange={this.onValueChange}
        value={this.state.switchValue}/>
    </View>
```


Announcements

- HW3 released. Closely related to today's lecture!
- Project 1 has been out for a week. Hopefully you've taken a look by now!
 - No new homework for Week 7 (next week) so you can focus on finishing Project 1 (due Week 8, i.e., 2 weeks from now)
- Final Project materials have been released!
 - Final Project Proposals due Nov. 4 (4 weeks from now). Start brainstorming!

Break Time!

A still from the movie 'Despicable Me' showing two yellow Minions in a break room. One Minion is standing and looking at a water cooler, while the other is bent over drinking from it. The room has green walls and a window in the background.

<<<<<<<<<<<<<<< App of the Week
>>>>>>>>>>>>>>>>

[illegible][illegible][illegible]

Some Annoying JavaScript Stuff

We can't go too much farther without dealing with a couple of JavaScript details that (IMHO) we shouldn't have to discuss at all. The first detail is the result of some poor choices (IMHO) that the JavaScript language designers made early on and then tried to fix later but ended up creating unnecessary confusion. The second detail is the result of an "improvement" to the language that (IMHO) nobody really needed all that much but made writing certain code patterns a bit easier and, some would say, more elegant. The result of both is that JavaScript coders take these things for granted and use them frequently in their code, with the result that it's almost impossible to find JavaScript tutorials, examples, or StackOverflow answers that don't assume knowledge of one or both.

Dealing with `this`

For these next two segments, we'll be working outside of React Native and back in plain old JavaScript + node. Switch to your `week6Sandbox` directory, fire up your editor, and create two files: `thiz.js` (intentionally misspelled) and `destruct.js`.

As you know, `this` refers to "the current object" when it appears within a class method, like so:

```
class Person {
  constructor(nm) {
    this.name = nm;
  }
  greet() {
    console.log("Hi, I'm " + this.name + "!");
  }
}

let joe = new Person("Joe");
joe.greet();
```

So far so good. But `this` doesn't always mean what you think it means. In particular, when a function is serving as an **event callback**, `this` does *not* refer to the "current object", but rather refers to the event that triggered the callback. If you add the following to the end of the above code

```
setTimeout(joe.greet, 500);
```

... you get the following output:

```
Hi, I'm Joe!
```

```
Hi, I'm undefined!
```

Where did "Joe" go? To dig a bit deeper, change the contents of `greet()` to

```
greet() {  
  //console.log("Hi, I'm " + this.name + "!");  
  console.log("In greet(), this is ", this);  
}
```

this 不再是object了，而是
event的触发者

And you will see the output:

```
In greet(), this is Person { name: 'Joe' }
```

```
In greet(), this is Timeout {
```

```
  _idleTimeout: 500,  
  _idlePrev: null,  
  _idleNext: null,  
  _idleStart: 77,  
  _onTimeout: [Function: greet],  
  _timerArgs: undefined,  
  _repeat: null,  
  _destroyed: false,  
  [Symbol(refed)]: true,  
  [Symbol(asyncId)]: 5,  
  [Symbol(triggerId)]: 1  
}
```

Kinda nutty. Now this nonsense with `this` would be just an idle curiosity, except in React Native we need to use event callbacks *all the time*, and what's more, we need to modify properties of `this` that belong to the containing object. Most specifically, we need to be able to modify `this.state` within a class that extends

`React.Component`, like so:

```
handleValueChange(value){
  this.setState({
    switchValue: value
  });
}
```

... which as we've just seen won't work, because `this` will be a `ValueChange` event, which won't have the method `setState()`. So what can we do?

Well, fortunately, when People Who Make JavaScript introduced the arrow function syntax in ES6 (2015), they fixed this little eccentricity—but only if you use arrow functions. To get back to Joe, you can fix the issue with `greet` simply by changing `greet` to be an arrow function:

```
class Person2 {
  constructor(nm) {
    this.name = nm;
  }
  greet = () => {
    console.log("Hi, I'm " + this.name + "!");
  }
}

let jen = new Person2("Jen");
jen.greet();
setTimeout(jen.greet, 500);
```

Works like a charm! Effectively, an arrow function inside of a class will bind `this` to the current object, as we would expect, instead of to the event.

The upshot is: ***when you have a class method that you want to use as an event callback, define it as an arrow function.***

There are other weirdnesses with `this` and other ways to fix the unexpected behavior of `this` when dealing with events (such as `function.bind()`), but we're

not going to get into those.

Destructuring

As you have probably noticed, JavaScript code typically makes heavy use of objects and properties. Objects are passed around and their properties are read and updated all the time. So often, in fact, that the People Who Make JavaScript decided to introduce a shorthand for getting at object properties a tiny bit easier, albeit at the expense of everyone who has to learn JavaScript and read other people's code. The shorthand is called "destructuring", and it works as follows. Imagine you have an object

```
let obj = {  
  prop1: "value 1",  
  prop2: "value 2"  
}
```

You could access `prop1` like so:

```
console.log("obj.prop1:", obj.prop1);
```

With *destructuring*, however, you can do this:

```
let {prop1} = obj;  
console.log("prop1", prop1);
```

What is happening here is you are creating a new variable (`prop1`), and assigning it the value of the property of *the same name* that is contained in `obj`. Note that the variable name inside of `{}` has to be exactly the same as the property name or it doesn't work.

Why bother? Honestly I'm not sure. Clearly it saves some typing if you're going to use `prop1` multiple times—each time you get to type just `prop1` instead of `obj.prop1`. You can also extract multiple properties at once, like so:

```
let { prop1, prop2 } = obj;
```

... which is cool I guess.

But whether I think it's useful or not, it's part of the language and people LOVE to use it, so you had better get used to it. Another way that you'll see this used is in

function signatures, as a way of extracting properties from objects that are expected to be passed to the function. So you might see something like this:

```
function proper({prop1}) {  
  console.log("in proper(), prop1:", prop1);  
}  
proper(obj);
```

... which means that `proper()` is expecting to be passed an argument which is an object that has a property called `prop1`. `proper()` doesn't care about the rest of the object, just `prop1`, so it cuts to the chase and *destructures* `prop1` right there in the parameter list. That function, by the way, would have the same behavior as this one:

```
function proper2(anObject) {  
  let prop1 = anObject.prop1;  
  console.log("in proper2(), prop1:", prop1);  
}  
proper2(obj);
```

You've already seen a destructuring statement. Last week when we first looked at `FlatLists`, we saw code like this:

```
<FlatList  
  data={this.fruit}  
  renderItem={({item}) =>  
    <View style={styles.itemContainer}>  
      <Text style={styles.itemText}>{item.type}</Text>  
    </View>  
  }  
>
```

And you're going to see more destructuring soon. You're going to encounter a function in the prep work you'll be assigned to do before next class, which will be to work through the first few steps of ["Getting Started" with React Navigation](#). In particular you'll see this:

```
function HomeScreen({ navigation }) {  
  return (  
    <View style={{ flex: 1, alignItems: 'center', justifyConte  
nt: 'center' }}>  
      <Text>Home Screen</Text>  
      <Button  
        title="Go to Details"  
        onPress={() => navigation.navigate('Details')}  
      />  
    </View>  
  );  
}
```

... which not only uses destructuring but also uses a function component, so we'll be taking a different approach. We will talk about what this all means in particular next week, but I wanted you to have some understanding of what you're looking at when you encounter this so it doesn't cause you any concern.

Text Input

Of course, you can only get so far with Switches. A few other React Native Components can handle more complex input (e.g., [Picker](#)), though somewhat surprisingly there aren't too many built-in components. Most UI components that you might like to use, such as Sliders, CheckBoxes, and (Radio)ButtonGroups are only available through 3rd party libraries like [React Native Elements](#). We're not going to cover these UI libraries in detail but I encourage you to check them out on your own at some point.

What we are going to cover is a particularly useful built-in component, `TextInput`. As you might surmise from the name, this is a component that lets you collect text input from the user.

To see `TextInput` in action, make a new project, and then edit your App.js to look like this:

```
import React from 'react';
import { StyleSheet, View, TextInput } from 'react-native';

export default class BasicInput extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <TextInput
          placeholder="Type here"
        />
      </View>
    )
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  }
});
```

Run the app and type in the box! Note that by default `TextInput` has no styling associated with it, so we had to use a placeholder so you could see where it is. As a side benefit, you now also know how to use a placeholder value for your `TextInput`.

As we did with the `Switch`, we now need to figure out how we can use the data that the user inputs in our program. Also, similarly to what we saw with the `Switch`, we will use an event handler to capture the data that the user inputs.


```
    <TextInput
      placeholder="Type here"
      onChangeText={textValue => this.setState({currentText: textValue})}
    />
```

Of course if you try this out now, you don't (or shouldn't) see any change—we're saving the user's input (maybe), but we can't really test to see if it's working as expected. To deal with this, we can add a `Text` label that displays the current value of `currentText`. Change `render()` to:

```
render() {
  return (
    <View style={styles.container}>
      <Text>{this.state.currentText}</Text>
      <TextInput
        placeholder="Type here"
        onChangeText={textValue => this.setState({currentText: textValue})}
      />
    </View>
  )
}
```

When you test the app, you'll get an error. How might you fix it?

Yes you need to make sure that `this.state.currentText` has an initial value so that something can be displayed when the app first runs (note: the "initial value" can be a blank string).

After fixing that, when you test the app, you should see that whatever is typed into the `TextInput` is mirrored exactly in the `Text` that is right above it.

Now You Try #3

Modify the code below to create an app that works like this:

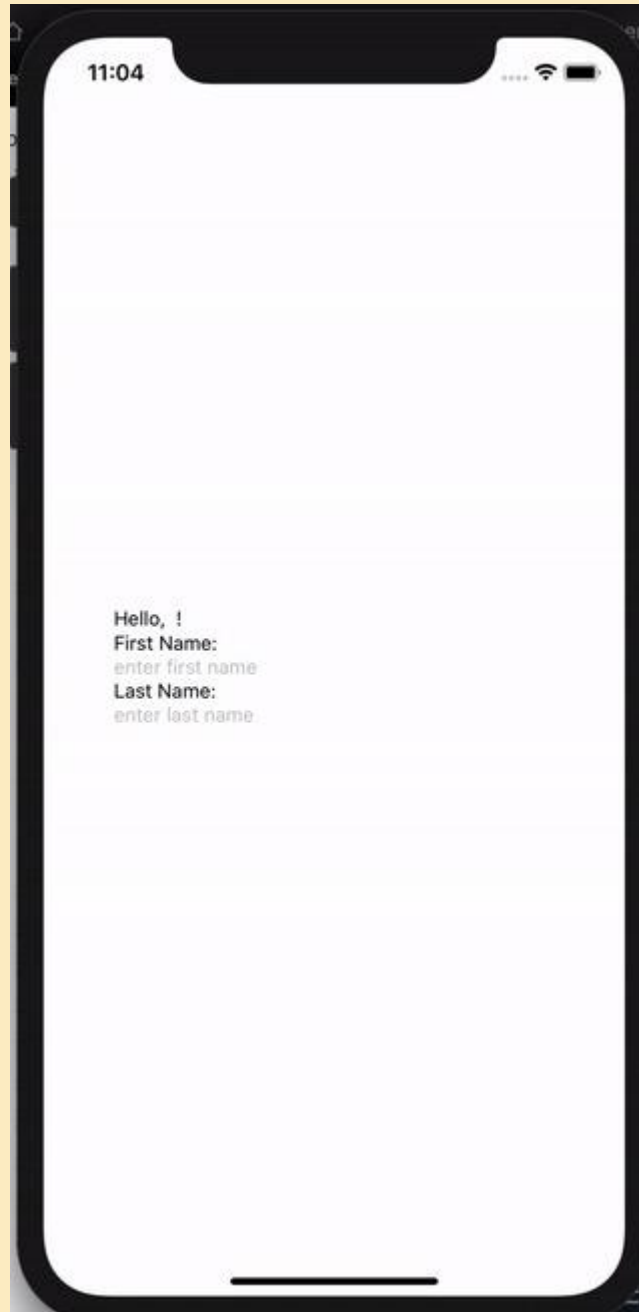


Figure out what code should replace each of the `/* Snippets */` and enter your solutions into [slido](#).

```
class App extends React.Component {  
  constructor(props) {
```

```
    super(props);
    /* Snippet #1: What goes here? */
}
render() {
    return (
        <View style={styles.container}>
            <Text>
                Hello, { /* Snippet #2: What goes here? */ }!
            </Text>
            <View>
                <Text>First Name: </Text>
                <TextInput
                    placeholder="enter first name"
                    onChangeText={ /* Snippet #3: What goes here? */ }
                />
            </View>
            <View>
                <Text>Last Name: </Text>
                <TextInput
                    placeholder="enter last name"
                    onChangeText={ /* Snippet #4: What goes here? */ }
                />
            </View>
        </View>
    )
}
}
```

Now You Try #4

Below you will find a solution to the above problem, partially rewritten to use a destructuring statement. Fill in the missing line of code to make this work!

Enter your solution into [slido](#).

```
class HelloName extends React.Component {

  constructor() {
    super();
    this.state = {
      firstName: "",
      lastName: ""
    };
  }
  render() {
    // Write a 'let' statement that uses destructuring
    // to assign the values of firstName and lastName correctly
    // for use in line 17 below
    return (
      <View style={styles.container}>
        <Text>
          Hello, {firstName} {lastName}!
        </Text>
        <View>
          <Text>First Name: </Text>
          <TextInput
            placeholder="enter first name"
            onChangeText={textValue => this.setState({firstName: textValue})}
          />
        </View>
      </View>
    );
  }
}
```

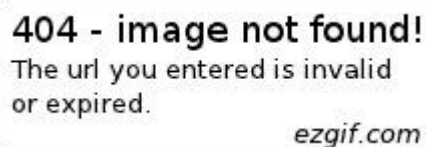
```

    </View>
    <View>
        <Text>Last Name: </Text>
        <TextInput
            placeholder="enter last name"
            onChangeText={textValue => this.setState({lastName: textValue})}
        />
    </View>
</View>
)
}
}

```

Now You Try #5: Bonus Challenge! (If there's time).

Write an app (either a new one, or modify `week6TextBox`) to work like the following. You can't see when the "Save" button is tapped, but the idea is that when you tap "Save", whatever is in the `TextInput` is "saved" to the `Text` field just above it.



No need to enter your code into slido (would be too much hassle to avoid slido's filters). Work on it til you're happy with it, and ask questions if you get stuck!

Today's Examples

- <https://github.com/Sl669-internal/week6Switch>
- <https://github.com/Sl669-internal/week6Sandbox>
- <https://github.com/Sl669-internal/week6TextInput>