1. **Hooks用法**
    A. **let [variable, updater] = useState(default)**
        a. **var是变量**
        b. **updater是更新var时调用的函数，可以接收new value或者function**
    B. **useEffect(Callable, notifier_list)**
        a. **Callable是被调用的函数**
        b. **Callable可以返回另一个函数，在组件unmount的时候作为cleaner**
        c. **当notifier_list中的变量被更新时，Callable会被调用**
        d. **如果没传入参数，则每次render会调用Callable**

08. Hooks, More Nav, More CRUD

CRUD Quick Review

React Hooks

What are Hooks? (And Why?)

Hooks were introduced by the React (js) team in 2018. They were introduced to solve problems that, for the most part, we don't have so far and aren't all that likely to encounter in 669. They do, however, make certain programming patterns easier to implement, at the cost of hiding a bunch of functionality behind some magic. Because of the magic, I've been reluctant to teach Hook in 669 (I don't like asking students to do things that I don't expect them to fully understand, and I don't fully understand how Hooks work behind the scenes—though I can make some pretty good guesses). However, Hooks have mostly taken over the React and React Native communities over the last 3 years and it's increasingly hard to find React code examples that don't use Hooks. Almost any 3rd party library you look into will use Hooks in their documentation, for example. As a result, we're going to learn about Hooks.

In a nutshell, Hooks provide a way to get the benefits of using class Components without using classes and, most importantly, without having to deal with the `this` keyword at all. Getting rid of classes and `this` helps to avoid a major source of confusion for developers without a deep background in OO programming, and it also cleans up the code to an extent, making code easier to write, read, and maintain. To take advantage of Hooks, you need to learn a few new things, though.

The `useState` Hook

Let's jump right in with an example. Modify your `week8Hooks` App.js to look like the following (leave `styles` intact):

```
import { StatusBar } from 'expo-status-bar';
import React, {useState} from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  let [greeting, setGreeting] = useState("Hello");
  创建greeting变量初始化为Hello，可以用setGreeting函数来更新它
  return (
    <View style={styles.container}>
      <Text>{greeting}</Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

And let's unpack the example.

Firstly, `useState` is imported from the `react` package, and then called at the beginning of the `App()` function. `useState()` returns an array with exactly two items, and this code (along with every example of React Hooks I've ever seen) uses *list destructuring* to assign the two returned items to the variables `greeting` and `setGreeting`. The array returned by `useState()` is always the same—the first element is a variable, which can be of any type (e.g., string, number, array, object), and the second element is a *function*. The first element, in this case `greeting`, is *your state variable*—that is, a variable that you want to use as part of your component's state. In this case the `greeting` returned by `useState()` is effectively the equivalent of `this.state.greeting` in a class component. The second element, `setGreeting`, is an *update function* that is used for changing the `greeting` state variable. It is, in essence, a `setState()` function that is *just for the one variable* (again, `greeting` in this case).

When you call `useState()`, you can (must?) pass an argument. This argument will be used to set the *initial value of the tracked variable.* Thus in the case above, `greeting` is initialized with the value `"Hello"`, which is the value it will have until it is changed by a call to `setGreeting()`.

To see how the updater function works, let's expand our example:

```
function App2() {
  let [greeting, setGreeting] = useState("Hello");
  let [counter, setCounter] = useState(0);

  return (
    <View style={styles.container}>
      <Text>{greeting}</Text>
      <Text>You've pressed the button {counter} times!</Text>
      <Button
        title="More!"
        onPress={()=>{
          setCounter(counter + 1);
        }}
      />
      <StatusBar style="auto" />
    </View>
  );
}
```

When you try this out, you should see the `{counter}` increment by one each time.
Note:

1. You no longer need to call `this.setState()`. Every updater function returned by `useState()` acts like `this.setState()` in that it changes the data model (i.e., `state`) *and* forces the UI to update.
2. You no longer need to initialize `state` in a constructor, or refer to `this.state` every time you want to get the value of a `state` variable.

Convenient! Why does it work? Magic! (Really it's not magic, but it's not something we're going to get into here.)

The `useEffect` Hook

So that’s how to deal with `state` using `useState()`. It’s worth repeating that Hooks are meant to be used in function components, not class components. While they might work in class components, it’s not how they’re intended. You should either use class components with `this.state` and `this.setState()` *or* use function components with Hooks, but mixing them together wouldn’t be a good idea.

So what about some of the other features of class components that you might need in particular cases? Last week we looked at *lifecycle hooks* for performing actions at particular points in the lifecycle of a `React.Component`, such as when it’s created and when it’s destroyed. Things you might want to do at such junctures include “expensive” operations like fetching piles of data, or initializing processes like timers or subscriptions that the Component will need to do its job. Since we can’t override `Component` methods like `componentDidMount()` when using a function component (since there are no methods to override), the authors of Hooks provided a catch-all solution in the `useEffect()` Hook, which allows you to define functionality that will run *whenever any lifecycle transition occurs in your component.* Again, let’s look at an example and then unpack how it works.

```
function App3() {
  let [greeting, setGreeting] = useState("Hello");
  let [counter, setCounter] = useState(0);

  useEffect(()=> {
    console.log("I'm using an effect!");
  });

  console.log("About to render");
  return (
    <View style={styles.container}>
      <Text>{greeting}</Text>
      <Text>You've pressed the button {counter} times!</Text>
      <Button
```

```
        title="More!"
        onPress={()=>{
          setCounter(counter + 1);
        }}
      />
      <StatusBar style="auto" />
    </View>
  );
}
```

Now You Try #1

Run the above example.

When does the function defined in `useEffect()` run? What can you conclude about when "effects" are triggered in the `App` component?

Answer in slido.

One-shot Effects

What you hopefully noticed is that an "effect" is triggered when the component first loads, and then every time you press the button. You may also have noticed that the function you provided to `useEffect()` is called *after* the function returns, i.e., after the JSX is provided to React Native (via the return), i.e., *after the component renders.* And in fact, this is exactly how the `useEffect()` hook is described in the Hooks documentation:

> ***Does `useEffect` run after every render?*** *Yes! By default, it runs both after the first render and after every update. (We will later talk about how to customize this.) Instead of thinking in terms of "mounting" and "updating", you might find it easier to think that effects happen "after render". React guarantees the DOM has been updated by the time it runs the effects.*

OK, so there you go. But what if you *don't* want an effect handler to run on every update? Say, for example, that you want to set up a subscription when the component mounts (i.e., after the first render), but setting up a new subscription on every update would be silly (at best—often this would be disastrous). The `useEffect()` Hook provides a solution to this, by allowing you to specify a *second* argument that specifies a *list of variables to check for changes*. If any of the variables in the list have changes since the last time `useEffect()` ran, it will run again. If none of the variables in the list have changes since last time, *`useEffect()` will not run.*

```
function App4() {
  let [greeting, setGreeting] = useState("Hello");
  let [counter, setCounter] = useState(0);

  const neverChange = "I will never change";

  useEffect(()=> {
    console.log("I'm using an effect!");
  }, [neverChange]);   每次neverChange变了才会运行useEffect

  console.log("about to render");

  return (
    <View style={styles.container}>
      <Text>{greeting}</Text>
      <Text>You've pressed the button {counter} times!</Text>
      <Button
        title="More!"
        onPress={()=>{
          setCounter(counter + 1);
        }}
      />
      <StatusBar style="auto" />
    </View>
  );
}
```

Fortunately you don't have to actually define a bogus variable that never changes to ensure that `useEffect()` only runs once, instead you can just pass an empty array as the second argument. Here's a slightly more realistic example of how you would use this pattern to set up a subscription in `useEffect()` and make sure it only runs once.

```
function App4() {
  let [greeting, setGreeting] = useState("Hello");
  let [counter, setCounter] = useState(0);

  useEffect(()=> {
    console.log("using effect");
    setInterval(()=>{
      setCounter(counter => counter + 1);
    }, 1000);
  }, []);

  return (
    <View style={styles.container}>
      <Text>{greeting}</Text>
      <Text>You've pressed the button {counter} times!</Text>
      <Button
        title="More!"
        onPress={()=>{
          setCounter(counter => counter + 1);
        }}
      />
      <StatusBar style="auto" />
    </View>
```

传入的是函数而不是新的值

```
  );
}
```

Here, we're using a slightly different version of the `setCounter()` updater function, where instead of passing the new value we pass a *function that takes the old value as a parameter and returns the new value.* As with `setState()`, both versions are permitted (just passing the new state or passing a function that receives the old state and returns the new state). The function version is somewhat "safer" in that it insulates you against whatever magic `setState()`/Hooks is doing under the hood to manage different version of your state variable. To be honest, I don't know why the function version is needed here other than that there are multiple layers of asynchrony going on (`useEffect()`, `setCounter()`, and `setInterval()` all involve asynchrony in one form or another), which can result in some messiness. It appears that supplying a function, at least in this case, works better than the alternative (`setCounter(counter + 1)`), which behaves pretty weirdly. Feel free to try it out.

Cleaning up Effects
When a Component unmounts, for example when you navigate away from it, any long-running processes or subscriptions that you initiated when the component mounted won't automatically be cancelled. In a class component, you typically clean these up by overriding `componentWillUnmount()`, but with Hooks we don't have all these lifecycle events—we just have `useEffect()`. Since unmounting is a special case, and can only happen once, the `useEffect()` hook provides a special mechanism for dealing with cleanup. The function that you provide to `useEffect()` can *return another function* that handles cleanup, and this function will only be run when the component is about to be destroyed. You won't always need to provide this function (that is, `useEffect()` will often return *nothing*, but if you ever need to do cleanup this is how you do it. In our example, we should cancel the interval timer when the component unmounts, otherwise the timer could run forever!

```
function App4() {
  let [greeting, setGreeting] = useState("Hello");
  let [counter, setCounter] = useState(0);

  useEffect(()=> {
    let intervalTimerID = setInterval(()=>{
      setCounter(counter => counter + 1);
    }, 1000);
    return (()=>{
      clearInterval(intervalTimerID);
    });
  }, []);

  return (
    <View style={styles.container}>
      <Text>{greeting}</Text>
      <Text>You've pressed the button {counter} times!</Text>
      <Button
        title="More!"
        onPress={()=>{
          setCounter(counter => counter + 1);
        }}
      />
      <StatusBar style="auto" />
    </View>
  );
}
```

返回的函数会作为cleaner在unmount的时候运行
传入TimerID以停止timer，否则navigator回去之后还会继续运行

Unfortunately there's no way I can figure out to test the cleanup function with a single-component app (i.e., an app with no navigation), so you'll just have to trust me for now.

Now You Try #2

1. In general, it's a bad idea to update state (in this case, by calling `setCounter()`) inside of a function that you provide to `useEffect()`. Why?
2. But we do it here anyway. Why does it not create the bad situation you (hopefully) identified in your answer to the first question?

因为可能会重新render

Answer in slido.

Checkpoint—everything we've done so far: https://github.com/SI669-internal/week8Hooks

Up Next

We'll be working with `week8NavWithCRUD`, which you hopefully created already. However, whether you did or not there are a few extra steps, which you should start now so that they'll be done before break:

```
$ expo init week8NavWithCRUD // if you didn't already...
$ cd week8NavWithCRUD
```

Next, install all of the navigation packages we'll need:

```
$ npm install @react-navigation/native
$ expo install react-native-screens react-native-safe-area-con
text
$ npm install @react-navigation/native-stack
```

And then install React Native Elements, which is a 3rd party UI library that offers significant improvements over the default React Native UI components.

```
$ npm install react-native-elements
```

More Nav, More CRUD

With Hooks under our belt, now we can return to our themes from last week, CRUD and Navigation. Our goal for today will be to put the two together. For the rest of today, we'll be working with `week8NavWithCRUD`, which you hopefully set up (by installing needed packages) before the break.

Set Up Navigation

Our plan with this example app is to define 2 screens, "Home" and "Details" that the user can navigate between. While this simple app could easily be defined in a single file (i.e., `App.js`), we're going to break it into multiple files for practice, since many apps that you will want to write in the future (even the near future) will quickly become unwieldy if you try to fit them into a single file.

As you will recall from last week, creating multi-screen apps requires

1. Setting up the routing structure (i.e., what routes can be navigated to, and what React Components are associated with each route)
2. Creating each of the Screens to which a user can navigate.

Set up routing in App.js

For today's example, we're going to use `App.js` *only* for setting up the routing, and define each screen (`HomeScreen` and `DetailsScreen`) in their own files. Here is our `App.js`:

```
import React from 'react';
import { NavigationContainer } from '@react-navigation/nativ
e';
import { createNativeStackNavigator } from '@react-navigation/
native-stack';
import HomeScreen from './HomeScreen';
import DetailsScreen from './DetailsScreen';


const Stack = createNativeStackNavigator();


function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator initialRouteName="Home">
        <Stack.Screen
            name="Home"
            component={HomeScreen}
            options={{ title: 'ToDo' }}/>
        <Stack.Screen name="Details" component={DetailsScreen}
/>
```

```
      </Stack.Navigator>
    </NavigationContainer>
  );
}

export default App;
```

A few things to note here:

- `const Stack = createNativeStackNavigator()` is run outside of any functions, so it is run immediately when `App.js` is processed during app startup, and before the JSX is processed. The purpose of this call is to create a `Stack` object that can be used within the `NavigationContainer`. Note that `Stack` is used within the JSX, but for this to work the `Stack` object needs to exist beforehand.
- We are importing and referencing components that we have not yet defined, as shown in **bold** above. Next we will define these components.

Create the `HomeScreen` to display the ToDo List

Create a new file in the project directory with the name `HomeScreen.js`. The contents will be:

```
import React, {useState} from 'react';
import { FlatList, StyleSheet, Text, View } from 'react-nativ
e';
import { Button, CheckBox, Input } from 'react-native-element
s';
import { MaterialIcons as Icon } from '@expo/vector-icons';

function HomeScreen({navigation}) {
  let initList = [
    {text: "Get Milk", key: "1"},
    {text: "Pick up dry cleaning", key: "2"},
    {text: "Pay rent", key: "3"}
  ];
  const [todoList, setTodoList] = useState(initList);

  return (
    <View style={styles.container}>
      <View style={styles.listContainer}>
        <FlatList
          contentContainerStyle={styles.listContentContainer}
          data={todoList}
          renderItem={({item})=>{
            return (
            <View style={styles.listItem}>
              <CheckBox/>
              <Text style={styles.listItemText}>{item.text}</T
ext>
              <View style={styles.listItemButtons}>
                <Button
                  icon={<Icon name="edit" size={24} color="dar
kgrey"/>}
                  type="clear"
                />
                <Button
                  icon={<Icon name="delete" size={24} color="d
arkgrey"/>}
                  type="clear"
                />
              </View>
            </View>
            );
          }}
        />
      </View>
      <Button
        title="Add Item"
        onPress={()=>{
          navigation.navigate("Details");
        }}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'flex-start',
  },
  listContainer: {
    flex: 0.5,
    padding: 30,
    width: '100%',
  },
  listContentContainer: {
    justifyContent: 'flex-start',
  },
  listItem: {
    flex: 1,
    justifyContent: 'flex-start',
    alignItems: 'center',
    flexDirection: 'row',
    padding: 5
  },
  listItemText: {
    flex: 0.7,
    fontSize: 18
  },
  listItemButtons: {
    flex: 0.2,
    flexDirection: 'row',
  }
});
export default HomeScreen;
```

Whew! That's a lot in one go! Let's unpack...

First the imports:

```
import React, {useState} from 'react';
import { FlatList, StyleSheet, Text, View } from 'react-nativ
e';
import { Button, CheckBox } from 'react-native-elements';
import { MaterialIcons as Icon } from '@expo/vector-icons';
```

Mostly what's new here is the use of React Native Elements UI components and the use of Icons from **expo icons**. We're not going to be able to examine either of these in detail, but know that React Native Elements UI components work pretty much like vanilla React Native UI Components, and to understand how they work in detail, you just need to read the docs. The same is true of Expo Icons (which you probably already figured out for Project 1).

Next the Component state, which we are now managing with Hooks. Here we have some dummy data, which is then used to initialize the `todoList` state variable. The `setTodoList` updater function is created at the same time.

```
function HomeScreen({navigation}) {
  let initList = [
    {text: "Get Milk", key: "1"},
    {text: "Pick up dry cleaning", key: "2"},
    {text: "Pay rent", key: "3"}
  ];
  const [todoList, setTodoList] = useState(initList);
```

Next we use JSX to define a UI that uses a `FlatList` to render all of the `todoList` items, along with some buttons for editing and deleting each item (associated with the item inside the `renderItem` function provided to the `FlatList`, and inactive for now) and a checkbox for marking todo items as "completed" (also inactive). Also, at the bottom of the screen we have an "Add Item" button that, for now, just navigates to the (yet to be defined) `DetailsScreen`.

```
  return (
    <View style={styles.container}>
      <View style={styles.listContainer}>
        <FlatList
          contentContainerStyle={styles.listContentContainer}
          data={todoList}
          renderItem={({item})=>{
            return (
            <View style={styles.listItem}>
              <CheckBox/>
              <Text style={styles.listItemText}>{item.text}</T
ext>
              <View style={styles.listItemButtons}>
                <Button
                  icon={<Icon name="edit" size={24} color="dar
kgrey"/>}
                  type="clear"
                />
                <Button
                  icon={<Icon name="delete" size={24} color="d
arkgrey"/>}
                  type="clear"
                />
              </View>
            </View>
            );
          }}
        />
      </View>

      <Button
        title="Add Item"
        onPress={()=>{
          navigation.navigate("Details");
        }}
      />
    </View>
  );
}
```

Additionally, in `HomeScreen.js` we have a `styles` object, which contains no surprises, but a lot of tedious details:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'flex-start',
  },
  listContainer: {
    flex: 0.5,
    padding: 30,
    width: '100%',
  },
  listContentContainer: {
    justifyContent: 'flex-start',
  },
  listItem: {
    flex: 1,
    justifyContent: 'flex-start',
    alignItems: 'center',
    flexDirection: 'row',
    padding: 5
  },
  listItemText: {
    flex: 0.7,
    fontSize: 18
  },
  listItemButtons: {
    flex: 0.2,
    flexDirection: 'row',
  }
});
```

And finally, we can’t forget to `export default` the `HomeScreen` function:

```
export default HomeScreen;
```

Create the `DetailsScreen` to display individual ToDo Items

In a new file called `DetailsScreen.js`, insert the following:

```
import React, {useState} from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { Input, Button } from 'react-native-elements';

function DetailsScreen({navigation, route}) {

  const [inputText, setInputText] = useState('');

  return (
    <View style={styles.container}>
      <View style={styles.inputArea}>
        <Text style={styles.inputLabel}>Item:</Text>
        <Input
          containerStyle={styles.inputBox}
          placeholder="New Todo Item"
          value={inputText}
          onChangeText={(text)=>setInputText(text)}
        />
      </View>
      <View style={styles.buttonArea}>
        <Button
          containerStyle={styles.button}
          title="Cancel"
          onPress={()=>{
            navigation.navigate("Home");
          }}
        />
        <Button
          containerStyle={styles.button}
          title="Add Item"
          onPress={()=>{
            navigation.navigate("Home");
          }}
        />
      </View>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'flex-start',
    padding: 30
  },
  inputArea: {
    flex: 0.1,
    flexDirection: 'row',
    justifyContent: 'flex-start',
    alignItems: 'center'
  },
  inputLabel: {
    flex: 0.2,
    textAlign: 'right',
    fontSize: 18,
    paddingRight: 10,
    paddingBottom: 10
  },
  inputBox: {
    flex: 0.8,
  },
  buttonArea: {
    flex: 0.1,
    flexDirection: 'row',
    paddingTop: 30,
    justifyContent: 'space-between',
    //alignItems: 'center',
    width: '70%',
    //backgroundColor: 'tan'
  },
  button: {
    width: '40%'
  }
});

export default DetailsScreen;
```

Unpacking, the imports are by now familiar—standard stuff with a couple of upgraded UI components:

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import { Input, Button } from 'react-native-elements';
```

The `DetailsScreen` component itself is just an `Input` (the React Native Elements version of a `TextInput`) and a couple of RNE `Buttons`, which for now just navigate back to the `HomeScreen`.

```
function DetailsScreen({navigation, route}) {
```

```
const [inputText, setInputText] = useState('');

return (
  <View style={styles.container}>
    <View style={styles.inputArea}>
      <Text style={styles.inputLabel}>Item:</Text>
      <Input
        containerStyle={styles.inputBox}
        placeholder="New Todo Item"
        value={inputText}
        onChangeText={(text)=>setInputText(text)}
      />
    </View>
    <View style={styles.buttonArea}>
      <Button
        containerStyle={styles.button}
        title="Cancel"
        onPress={()=>{
          navigation.navigate("Home");
        }}
      />
      <Button
        containerStyle={styles.button}
        title="Add Item"
        onPress={()=>{
          navigation.navigate("Home");
        }}
      />
    </View>
  </View>
  );
}
```

And then, of course, some styling:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'flex-start',
    padding: 30
  },
  inputArea: {
    flex: 0.1,
    flexDirection: 'row',
    justifyContent: 'flex-start',
    alignItems: 'center'
  },
  inputLabel: {
    flex: 0.2,
    textAlign: 'right',
    fontSize: 18,
    paddingRight: 10,
    paddingBottom: 10
  },
  inputBox: {
    flex: 0.8,
  },
  buttonArea: {
    flex: 0.1,
    flexDirection: 'row',
    paddingTop: 30,
    justifyContent: 'space-between',
    //alignItems: 'center',
    width: '70%',
    //backgroundColor: 'tan'
  },
  button: {
    width: '40%'
  }
});
```

... and the export at the end:

```
export default DetailsScreen;
```

At this point, everything should be wired up and ready to test. Give it a try.

Defining a DataModel

At this point, we've defined our navigation structure in `App.js` and our two `Screens`: the `HomeScreen` to display our entire ToDo list, and the `DetailsScreen` to display individual ToDo items. These elements represent our application's **View** (the display of ToDo list items) and **Controller** (operations on our ToDo list, including adding, editing, and deleting items—though these don't work yet). But where is our **Model**?

Note: in the remainder of this lesson I will walk through but one of MANY ways to handle `state` that is shared across several components. This is such a common issue, that there exist very popular packages such as Redux for managing application state. Redux and its kin are very useful and VERY popular—so much so that Redux is almost considered an integral part of React Native, as it is so commonly used in React Native apps. While Redux (and others like it) are very powerful and useful, the initial learning curve is somewhat steep and we won't have time to cover it in 669. The approach I will cover here has a fair amount of conceptual affinity with the Redux approach, but it's simpler (if you can believe it) and doesn't require learning any fundamentally new concepts or programming patterns. I think if you can gain comfort with the “Data Model” approach we will use here, you will be well poised to adopt more industrial strength state-management approaches like Redux in the future.

Component State vs Application State

We have some elements of our Model defined in the form of our state variables. These include `todoList` in the `HomeScreen` and `inputText` in the `DetailsScreen`. In both cases, these variables contain crucial state that defines what is displayed and captured in their respective `View`s.

Properly construed, however, these state variables are really just *Component State*, in that they represent what is being displayed on their respective Components. We are missing an important element, which is the *Application State,* which represents the data that is *shared* across components—in this case that would be the underlying `todoList` that is being added to, updated, and deleted from by various operations spread across both of our screens. While we do have a `todoList` object (specifically an array) that is currently part of the `HomeScreen`, it will be better in the long run to separate this from any specific component and make it part of the app as a whole. To do this, we’ll create yet another file, which we’ll name `DataModel.js`:

```
// utility function for getting unique keys
// will go away when we start using persistent storage
let nextKey = 1;
function getNextKey() {
  return '' + nextKey++;
}

class DataModel {
  constructor() {
    this.todoList = [];

    //hardcoded list for testing
    this.todoList.push({text: "Get milk", key: getNextKey()});
    this.todoList.push({text: "Pick up dry cleaning", key: getNextKey()});
    this.todoList.push({text: "Pay rent", key: getNextKey()});

  }

  addItem(item) {
    item.key = getNextKey();
    this.todoList.push(item);
  }

  deleteItem(key) {
    let idx = this.todoList.findIndex((elem)=>elem.key===key);
    this.todoList.splice(idx, 1);
  }

  updateItem(key, newItem) {
    let idx = this.todoList.findIndex((elem)=>elem.key===item.key);
    this.todoList[key] = newItem;
  }

  getItem(key) {
    let idx = this.todoList.findIndex((elem)=>elem.key===item.key);
    return(this.todoList[key]);
  }

  getTodoList() {
    return this.todoList;
  }
  getTodoListCopy() {
    return Array.from(this.todoList);
  }
}

let theDataModel;

export function getDataModel() {
  if (!theDataModel) {
    theDataModel = new DataModel();
  }
  return theDataModel;
}
```

As usual, let’s unpack!

```
// utility function for getting unique keys
// will go away when we start using persistent storage
let nextKey = 1;
function getNextKey() {
  return '' + nextKey++;
}
```

As the comment says, this is just a helper function that we can count on to generate unique keys for each element in our ToDo list. Nothing fancy, but that’s OK because we aren’t going to be in the business of generating our own keys for long—usually we’ll count on our persistent storage (e.g., our backend database) to assign unique keys to our data elements. This is just a stopgap until we get there.

```
class DataModel {
  constructor() {
    this.todoList = [];
    //hardcoded list for testing
    this.todoList.push({text: "Get milk", key: getNextKey()});
    this.todoList.push({text: "Pick up dry cleaning", key: get
NextKey()});
    this.todoList.push({text: "Pay rent", key: getNextKey()});

  }
```

As we shall soon see, our `DataModel` consists not only of the data, which in this case so far is just an array called `todoList`, but also the operations to manipulate (create, read, update, delete) that data. To associate the application data and its related operations we'll use a class, here called simply "`DataModel`". In the constructor we initialize the `todoList` array and, for now, pre-populate it with some hardcoded values.

Next, we define the core operations that can be performed on our data. Note that none of these operations are defined in terms of any kind of UI. Indeed there is not even a trace of React Native anywhere in the entire `DataModel.js` file! Here, we're just using vanilla JavaScript code to manipulate a vanilla JavaScript array:

```
df


  deleteItem(key) {
    let idx = this.todoList.findIndex((elem)=>elem.key===key);
    this.todoList.splice(idx, 1);
  }


  updateItem(key, newItem) {
    let idx = this.todoList.findIndex((elem)=>elem.key===key);
    this.todoList[key] = newItem;
  }


  getItem(key) {
    let idx = this.todoList.findIndex((elem)=>elem.key===key);
    return(this.todoList[key]);
  }

  getTodoList() {
    return this.todoList;
  }

  getTodoListCopy() {
    return Array.from(this.todoList);
  }
```

Now You Try #3

1. Explain in your own words what each of the following `DataModel` methods are doing:
   a. `addItem()`
   b. `deleteItem()`
   c. `updateItem()`
   d. `getItem()`
   e. `getTodoList()`
   f. `getTodoListCopy()` (for this you might need to look up the docs for `Array.from()`)

Answer in slido.

The final bit of the Data Model is about exporting the model so that other parts of the application can use it. Here, we're implementing a software architecture pattern called the "Singleton Pattern," which is used whenever you want to make sure there is at most ONE instance of a particular class or data type within an application. Here the Singleton Pattern makes sense because we want to make sure that any Components that are updating the Data Model are updating the *same* Data Model. I'm sure you can imagine what kind of bad things could happen if each component had their own copy to modify and these changes weren't shared among the Components.

```
let theDataModel;


export function getDataModel() {
  if (!theDataModel) {
    theDataModel = new DataModel();
  }
  return theDataModel;
}
```

Create: Adding ToDo List Items

Adding an item requires two steps:

1. On the `HomeScreen`, click the “Add Item” button, which will trigger navigation to the `DetailsScreen`
2. On the `DetailsScreen`, enter text for the new item and click the “Add Item” on that screen.
3. Navigate back to the `HomeScreen`, with the new item added to the list.

Some of this is already working, including:

1. Navigating to the `DetailsScreen` when “Add Item” is clicked:

In `HomeScreen.js`:

```
      <Button
        title="Add Item"
        onPress={()=>{
          navigation.navigate("Details");
        }}
      />
```

2. Capturing user input in the `DetailsScreen` `TextInput`:

```
  const [inputText, setInputText] = useState(itemText);


...
```

```
        <Input
          containerStyle={styles.inputBox}
          placeholder="New Todo Item"
          value={inputText}
          onChangeText={(text)=>setInputText(text)}
        />
```

3. Navigating back to the `HomeScreen` when the user clicks “Add Item”, and also when they click “Cancel”. In the latter case, of course, the navigation should happen but no changes to the Data Model should be registered:

```
      <View style={styles.buttonArea}>
        <Button
          containerStyle={styles.button}
          title="Cancel"
          onPress={()=>{
            navigation.navigate("Home");
          }}
        />
        <Button
          containerStyle={styles.button}
          title="Add Item"
          onPress={()=>{
            navigation.navigate("Home");
          }}
        />
      </View>
```

4. Adding an item to the Data Model:

```
   addItem(item) {
    item.key = getNextKey();
    this.todoList.push(item);
  }
```

Mostly we need to glue this all together, which we can do by

1. Deriving the `HomeScreen`'s todoList state variable from the `DataModel`.

```
// ... other imports
import { getDataModel } from './DataModel';

function HomeScreen({navigation}) {
~~  // let initList = [~~
~~  //     {text: "Get Milk", key: "1"},~~
~~  //     {text: "Pick up dry cleaning", key: "2"},~~
~~  //     {text: "Pay rent", key: "3"}~~
~~  // ];~~
~~  // const [todoList, setTodoList] = useState(initList);~~

  const dataModel = getDataModel();
  const [todoList, setTodoList] = useState(dataModel.getTodoLi
stCopy());
```

2. Providing the `DetailsScreen` with an instance of the `DataModel` as well.

```
// ... other imports
import { getDataModel } from './DataModel';

function DetailsScreen({navigation, route}) {
  const [inputText, setInputText] = useState('');
  const dataModel = getDataModel();
```

3. And then using the `DataModel` in the `onPress` handler for the `DetailsScreen`'s "Add Item" button:

```
        <Button
          containerStyle={styles.button}
          title="Add Item"
          onPress={()=>{
            // update data model
            dataModel.addItem({text: inputText});
            console.log('new data model: ', dataModel.getTodoL
ist());
            navigation.navigate("Home");
          }}
        />
```
Pretty simple, eh? Try this out.

Now You Try #4

Note the `console.log()` statement in the `onPress` handler for “Add Item”. Compare the console output after you add an item with what you see when you get back to the `HomeScreen`. What, if anything is different? What do you think is going on? How do you think you would fix the problem you see?

Answer in slido.

Subscribing to DataModel updates

You probably noticed that the data model changes showing in the console were not showing up on the `HomeScreen`. You may have further surmised that this was happening because nothing was happening in the code that was forcing the `HomeScreen` to update its UI. Even though the change was being made to the underlying `DataModel`, and even though the `HomeScreen` had a copy of the `DataModel`, the `HomeScreen` just sat there, blissfully unaware of the change to the data it was supposed to be showing.

In a single screen, this is an easy problem. Simply call `setState()` when the data changes, and the UI updates are taken care of. With Hooks, simply call `updaterFunction()` (e.g., `setTodoLIst()`, `setCounter()`) and all is well.

When state is shared among components it's not so easy. Unfortunately, one component (say, `DetailsScreen`) can't simply call `setState()` on another component (say, `HomeScreen`). The reasons for this are a bit complicated, but suffice it to say that React Native doesn't allow this kind of loosey goosey interaction among components. Moreover, while components can pass (mostly simple) data to other components when they navigate to them in a "forward" direction, it's not so easy to pass data "back" down the stack. More importantly, even if you can pass data "back" (and actually you can), it's not easy to get the receiving component to *notice* the change in the passed data, since components that are lower in the stack aren't unmounted when you navigate to a new component (i.e., push a new component onto the stack), nor are they re-mounted when you navigate back (i.e., pop a component off the stack). The "lower" component just sits there, thinking it's being displayed, until the other component goes away. There are no lifecycle events (e.g., `componentDidMount()` or `componentWillUnmount()`) that we can rely on and intercept to force an update. As a result, we need to create our own events, in a sense.

底层组件不知道被覆盖了

To "create our own events," we will augment our DataModel to support "subscriptions," which will allow Components to subscribe to be notified whenever the DataModel's data changes. There are a number of programming patterns that support the basic idea of subscribing for updates, such as the "Listener Pattern," the "Observer Pattern," and "Publish-Subscribe" (often called Pub-Sub). Each of these has subtle differences, and for our purposes we can get by with a simplified version that will be adequate for small-to-medium-sized apps. Our subscription approach will consist of the following:

1. Augment the `DataModel` to allow clients to subscribe to updates. Clients will provide a callback function that will be called whenever the `DataModel` has an update to share.

```
class DataModel {
  constructor() {
    this.todoList = [];
    this.subscribers = [];
    // ...
  }

  subscribeToUpdates(callback) {
    this.subscribers.push(callback);
  }
```

2. Modify `HomeScreen` to subscribe to updates upon creation, by taking advantage of the `useEffect()` Hook:

```
function HomeScreen({navigation}) {
  const dataModel = getDataModel();
  const [todoList, setTodoList] = useState(dataModel.getTodoLi
stCopy());

  useEffect(()=>{
    dataModel.subscribeToUpdates(()=>{
      setTodoList(dataModel.getTodoListCopy());
    });
  }, []);
```

Note that we are using *copies* of the `dataModel`'s `todoList` here. This is to insulate us against sudden changes in the data in the midst of a UI update (recall the issues with asynchrony and data updates), as well as to remind us that the component state is just a *shadow* of the application state. This will keep us honest and force us to be explicit about changes in each direction, which will help to prevent errors in the long run.

3. Finally, we need to change the `DataModel` to actually broadcast the updates when they happen, which includes implementing `updateSubscribers()` as well as calling it whenever data changes. When does data change? Why, whenever CRUD happens!

```
class DataModel {
  // ...
  subscribeToUpdates(callback) {
    this.subscribers.push(callback);
  }
  updateSubscribers() {
    for (let sub of this.subscribers) {
      sub(); // just tell them there's an update
    }
  }
  addItem(item) {
    item.key = getNextKey();
    this.todoList.push(item);
    this.updateSubscribers();
  }
  deleteItem(key) {
    let idx = this.todoList.findIndex((elem)=>elem.key===key);
    this.todoList.splice(idx, 1);
    this.updateSubscribers();
  }
  updateItem(key, newItem) {
    let idx = this.todoList.findIndex((elem)=>elem.key===item.
key);
    this.todoList[key] = newItem;
    this.updateSubscribers();
  }
  // ...
}
```

Now You Try #5

How would delete work? Here's a hint: everything you need to do can be done in the `onPress` handler for the "delete" button in the `HomeScreen`'s `FlatList`.

Give it a try, and submit your solution for the code to replace /* what goes here? */ into slido.

```
        <Button
              icon={<Icon name="delete" size={24}
color="darkgrey"/>}
              type="clear"
              onPress={()=>{
                   /* what goes here? */
              }}
        />
```

A solution, including delete *and update* can be found here: https://github.com/SI669-internal/week8NavWithCRUD