重点：

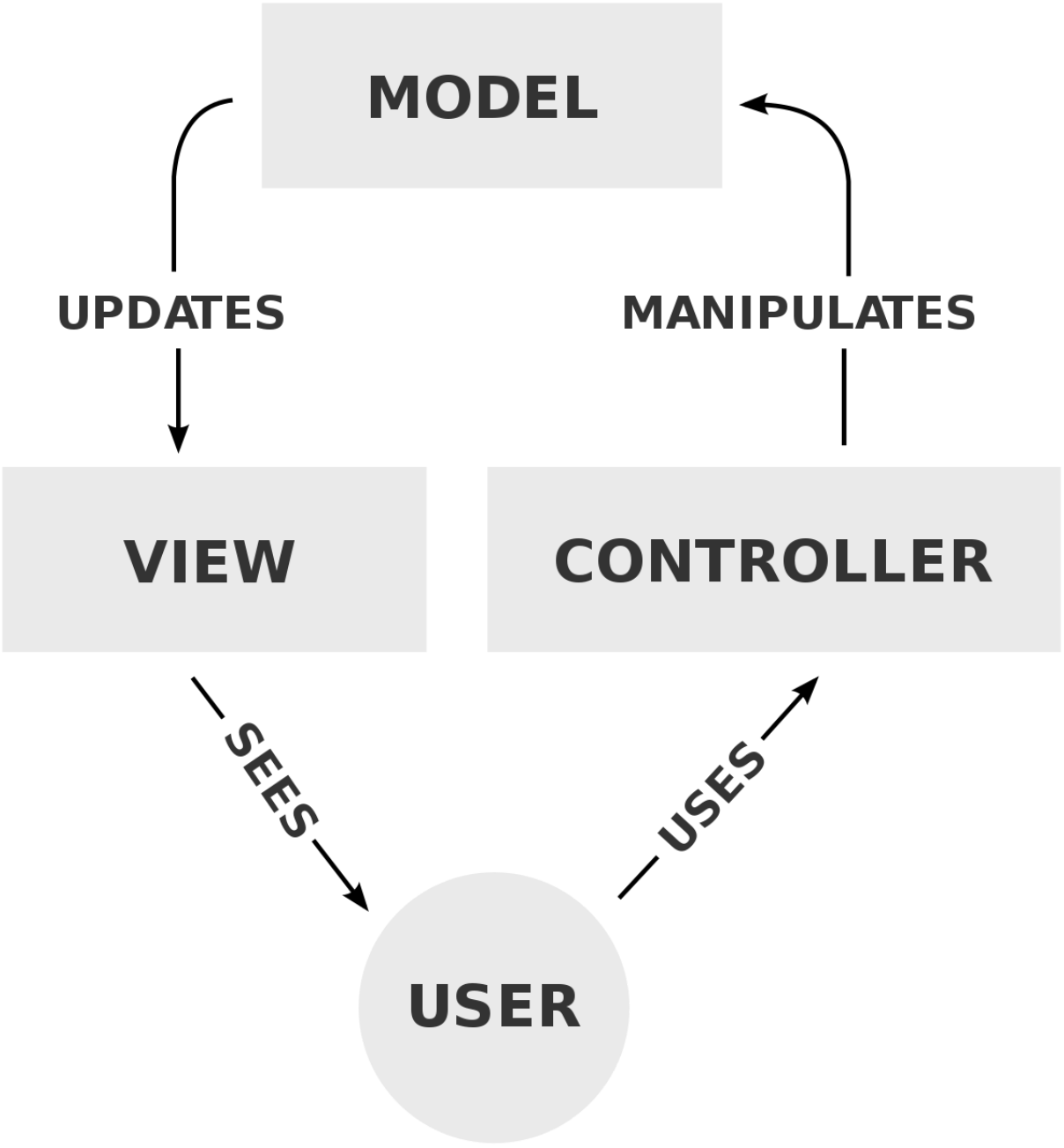
1. MVC（模型，界面，逻辑分离）
2. JSX中{}中放javascript 代码
3. 继承React.Component重载render方法实现界面渲染
    A. constructor中添加super()初始化父类
    B. this.state必须初始化
    C. this.setState({new state})修改state
    D. render运行时，不能直接调用回调函数，否则无限循环

04. Connecting Logic and the UI

The Model-View-Controller architectural pattern

Very early in the history of graphical user interfaces (like, in the late 70s) it was recognized that there was value in **separating the logic of the program** (how it behaves) **from its presentation to the user** (how it looks). For many programs, there are additional benefits in **separating the data model** (a representation of the data the program manipulates as well as that data's current state). As we will see shortly, separating these parts of the program has several benefits, mostly relating to the more general software engineering principle of ***separation of concerns.***

The most well-known architecture that supports the separation of program logic, presentation, and data model is called the **Model-View-Controller (MVC)** framework. Here is how the framework is usually depicted:



The MVC Framework

It is important to recognize that MVC is a **conceptual model, not necessarily a concrete description** of how a program's code is structured. With regards to the "View" and "Controller" components in particular, it can be hard to isolate code that presents information and code that allows users to manipulate that information and most of the time there isn't much benefit in doing so. However, being aware of the thinking behind an architecture like MVC (or its cousins MVP, MVVM, and VIPER)

can be very helpful for writing robust, testable, and maintainable applications. It can also be helpful for understanding the architectural decisions behind most of the modern application development frameworks you will use, including React Native.

The components of MVC are as follows:

Model 模型就是定义数据的方式，例如list或者dict等数据结构

An application's *model* refers to the **in-code data structures for representing the data an application uses** (e.g., lists, dictionaries, classes, objects) and the **operations provided in the code for manipulating it** (e.g., create, read, update delete). Even for applications that don't claim to use an MVC or any particular architecture you will often hear developers refer to the "data model." Data models can be as simple as a list (e.g,. the items in a todo list) or a few variables (e.g., the current score of a game) or as complex as multiple federated relational databases containing dozens of tables and billions of data entities. It's worth noting, though, that regardless of how the data is stored (only in program variables, in a local database, or on a cloud server), the term *data model* primarily refers to how that data is represented and accessed in the program code.

View

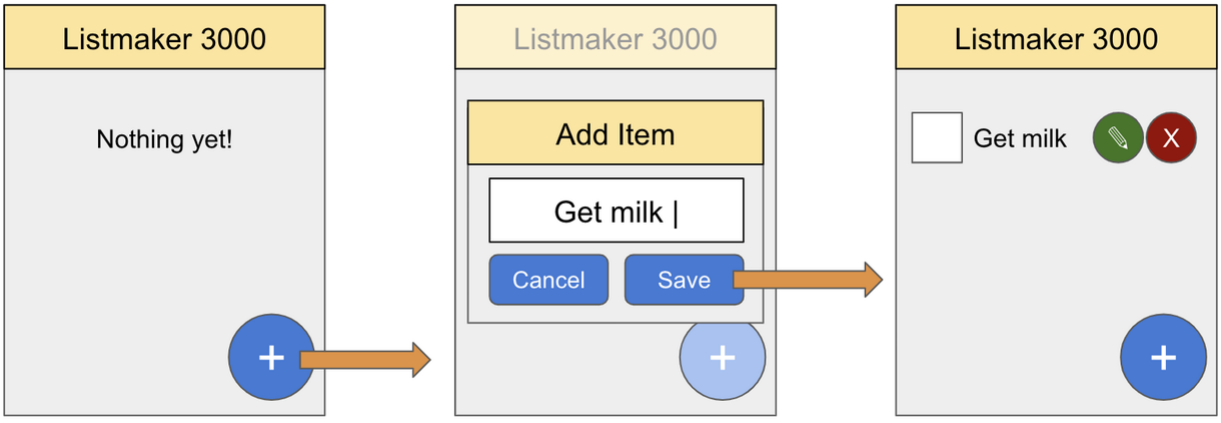
An application's *view* is what the user sees, or more specifically ***how application data is presented to the user.*** The view would include things like text fields, scrolling lists, tables, graphs, maps, icons, and so on. It would also include the overall organization and layout of the various individual elements of the view (labels, icons, frames, sections, etc.).

Controller

In a narrow sense, the "controller" component of an application is the part that **receives user input and enacts user commands** by manipulating the data model and, through the model, the view. However, it would be just as useful to think of the controller as "**everything in the program that isn't the view or the model**." Nowadays there can be lots of other things besides user commands that impact the logic of a program (broadly speaking we can think of the "logic" as defining *what* happens *when*). One such example would be network events—things happening on other devices (e.g., other users' activities such as messaging, commenting, or "liking") may well have an impact on the data and view on a particular device. Other examples of controller events not triggered by user input would include timers,

sensor signals, and push notifications (i.e., notifications generated by a server and pushed to a user's device).

Let's look at an example that will help to explain these concepts. Imagine a simple todolist app. You can add items by clicking the "+". You can edit and delete stuff too, but that isn't shown here.



Now You Try #1
1. Just by looking at the storyboard above, name one or more parts of the UI make up the *View.* 2. Which parts make up the *Controller*? 3. Where is the *Model*?
Answer on slido.

In this example, the main part of the UI that makes up the *View* is the list of todo items. In the first screen, there are no items so the view communicates that by saying "Nothing yet!" Once there is an item ("Get milk"), the list view displays that information to the user in such a way that it is clear that "Get milk" is a list item.

The UI elements that make up the *Controller* are the buttons that allow the user to manipulate the *Model*, aka the application data. These elements include the "+" button, the save and cancel buttons, the checkbox, and the buttons that allow editing and deletion of list items.

One could argue that the input box in the second panel provides both a *View,* in that it shows the current state of the temporary variable that represents the list item text that the user is inputting, and a *Controller* in that it allows the user to change that value. This example illustrates how pointless it is to get really nitpicky about what precisely is a View and what is a Controller, especially when it comes to direct manipulation interfaces.

What about the *Model*? You might have said that the one-item list that is showing on the screen in the third panel is the model. But it's not! What you can see in the user interface, by definition, is a *View* of the model. The model here is indeed the list, but it's the *data* for the list. The model code probably looks something like this:

```
let todoList = [];

function addItem(itemText) {
  todoList.push(itemText);
}

function removeItem(itemIndex) {
  todoList = todoList.splice(itemIndex, 0);
}

/// and so forth
```

That is, it consists of a data structure (in this case, an array) and a set of operations to manipulate the data structure. Of course, even with this simple example the data model is probably slightly more complex, like so:

```
class ListItem {
  constructor(itemText, checked=false) {
    this.itemText = itemText;
    this.checked = checked;
  }
```

```
  toggle() {
    this.checked = !this.checked;
  }
}

let todoList = [];

function addItem(itemText) {
  todoList.push(new ListItem(itemText));
}

// ... and so forth
```

What's important about this extended example is the that the observation that the *Model* is generally not one thing. All of the following are part of the application's *Model:*

- the definition of the `ListItem` class
- `todoList`
- the `addItem()` function

If you want to talk about the particular data that an application is working with at a particular time, you could refer to the ***state of the model*** or, more generally, the ***application state.*** In panel 3 of this example, the state of the model/application state (according to the code definitions provided above) is something like:

```
todoList = [ {
  itemText = "Get milk",
  checked = false,
} ];
```

That is, the application is working with an array `todoList` that has one item of type `ListItem` which has two properties: `itemText = "Get milk"` and `checked =`

`false`. As the program continues to execute, we would expect the state of the model to change, but the structure would not change—it would always be an array of `ListItems` with the same properties, though the properties would probably contain different values.

Separation of Concerns

As noted MVC was invented to provide several benefits. The benefits mostly derive from MVC's ability to support the notion of *separation of concerns.*

Most software of any significance is produced by teams of people. Making sure that different team members can work in parallel can be challenging. First, of course, it's important to make sure they don't make changes that conflict with each other. The usual solution to this is to have different people work on different parts of the code. This creates a second challenge, which is how to make sure that the different parts all fit together once everyone has finished their bit. The concept of *separation of concerns* describes a desirable property for software designs that support different people working on different things that ultimately need to be integrated. Each part of the system can be said to have its own *concerns,* and the more you can keep them separate, the better it will be for working in parallel.

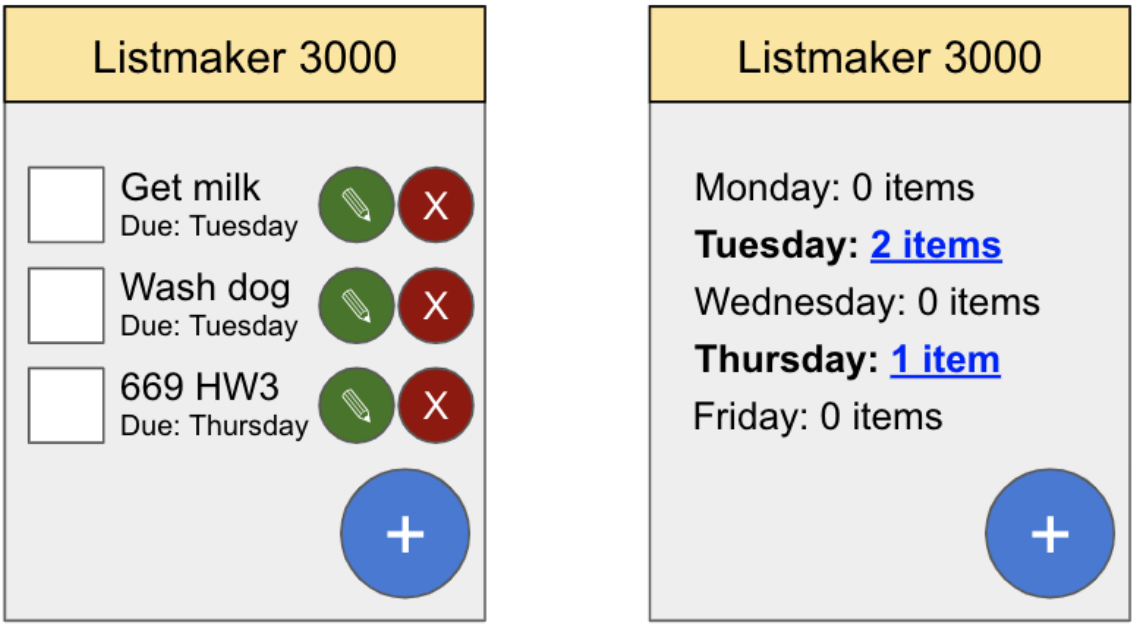
Separation of concerns provides benefits in a couple of different ways

Specialization: for many applications, you want to have team members with different skills. For example, you might have snazzy UI coders, graphic designers, database experts, backend architects, and so forth. You don't necessarily want your graphic designers writing backend code, and you don't want your database people creating the UIs. Separation of concerns allows each specialty to focus on what they're good at and ignore the stuff they don't understand or enjoy while still all contributing to the success of the project.

Decoupling: by keeping different parts of the application isolated from each other, it's possible for the different parts to evolve at different rates. You might, for example, start out with a very basic data model that stores data in files and can't handle more than a couple hundred data items without crashing the app. Such a placeholder model may be perfectly find for building an initial prototype and testing out the UI code. If the interface between the data model and the other parts of the system (e.g., the View) is well-designed, however, it should be possible to replace the

placeholder data model with one that is cloud-based, scalable, secure, lightning fast, etc. etc. while not changing the UI code at all. Thus even on small teams (even teams of one) that are not highly specialized separation of concerns can be beneficial because it allows the team to focus on a small set of issues at any one time and defer other issues until a time that they are better equipped to deal with them.

Reuse: Many applications need the same data and/or functionality in multiple places within the code. Thinking carefully about which pieces of code might need to be used in multiple places and then designing those pieces to facilitate re-use (i.e., write once and use in several places) can reduce errors, enhance maintainability, and reduce development effort. As an example, many applications may provide multiple views of the same data. Consider an alternative version of our todo list that has added support for scheduling todo items. It could provide two very different views of the exactly the same data.



Same model, same application state, different views, different controllers(?)

As you can imagine, these two views could be rendered from *exactly* the same data—meaning that they could share a data model (and the application state). However, they do *not* share View code—the way they render the data is completely different. This is an example of how the *Model* (including not only the data structures but also the operations for, e.g., adding, deleting, and editing items) could be re-used by multiple parts of the application (in this case different screens).

Template Languages

In modern web-based application development, a major way that separation of concerns manifests is in the separation of ***layout* from *behavior*** (we'll leave *style* out of this for the moment). In fact, these concerns are so separated that **entirely different languages are used for each concern**, with HTML being used for layout and JavaScript being used for behavior. Each of these languages is tuned for the purpose it serves. HTML is good for *declaring* the elements of a graphical layout in a way that expresses their structure (e.g., nesting tags inside other tags), type (e.g., `<h1>` vs `<img>`), and certain visual parameters (e.g., `<img src="puppies.jpg"/>` or `<table border=1>`). HTML is *terrible* at specifying interactive behaviors such as what should happen when a clicks a button or enters a search term. Indeed all HTML can do in such cases is redirect to another page (using `<a>`), but it can't really do anything else. A procedural language such as JavaScript, on the other hand, is *really good* at specifying behaviors but pretty darned bad at describing layouts (it can be done, but it's verbose and clunky and the organization of the code doesn't convey the organization of the user interface in any useful way).

In vanilla JavaScript, these concerns are integrated by embedding `<script>` tags into HTML documents, which allow the layout code (HTML) and the behavioral code (JS) to live side by side. Very small *hooks* such as event handlers provide the interface between these two types of code, as in this example from Lesson 02:

```
<head>
  <meta charset="utf-8">
  <title>Buttons</title>
</head>
<body>
  <script>
    function a() {
        document.getElementById("display").innerHTML = "A";
    }
    function b() {
        document.getElementById("display").innerHTML = "B";
    }
```

```
    console.log("JavaScript is alive."); //
  </script>

  <h1 id="display">   </h1>
  <button type="button" onclick="a()">Display A</button>
  <button type="button" onclick="b()">Display B</button>

</body>
</html>
```

Here the JavaScript is contained within the `<script>` tag and everything else is HTML, with the *interface* between the two being the two `onclick` statements within the `<button>` tags.

There is another way.

Many modern web-oriented frameworks employ *templates* as a way to more seamlessly integrate the UI layout (the View) with dynamic data that is generated and manipulated by the application's programmed logic (the Controller and the Model). A template is first and foremost a layout definition. It defines the various aspects of the UI and their relationship to each other. It also defines static (i.e., unchanging) aspects of the UI, such as headers, footers, navigation options, menus, and buttons. But templates start to get really interesting (and different from regular old HTML) is when they integrate variable data and snippets of code to "fill in" parts of the UI as the application is running.

A common form of templating involves creating HTML with embedded commands that use special syntax to either control (e.g., show or hide) the elements that are displayed or insert dynamic data that is determined at runtime to fill in "holes" in the UI. Here's an example of a JINJA2 template, such as you might have seen if you've worked with Flask:

```
<html>
    <head>
        <title>{{ title }}</title>
```

```
    </head>
    <body>
        <ul>
        {% for member in members: %}
          <li>{{ member }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

React and React Native have a template language too, called JSX, which is similar in some ways but quite different in others from more standard approaches like JINJA2. Let's look at an example:

```
export default function App() {
  let num = 9;
  return (
    <View style={{
        backgroundColor: 'fff',
        top: 200,
        left: 100,
        width: 200 }}>
      <Text style>Your lucky number:</Text>
      <Text style>{num}</Text>
    </View>
  );
}
```

花括号内是Javascript 代码

In line 7, you can see the expression `{num}`. The curly braces tell JSX that whatever is inside of them is JavaScript code. In this case, the code is just the name of a variable `num` that was defined earlier in the code. When this template is processed, the

contents of `{num}` will be determined by executing the code inside, and the result will be inserted into the template at that position.

Now You Try #2

1. Replace the `function App()` in your **week4App**'s `App.js` with the above `function App()`. Be sure to include `export default` and all that. Run the app using `expo start` and view it on your device or in a simulator.
2. Change the value of `num` and reload the running app to test it out (just saving the changed file should be enough to force a reload).
3. (Almost) Any JavaScript *expression* can go inside the `{}`. Replace `num` inside the brackets with `Math.round(Math.random() * 10)`. Test it out. Reload a couple of times to test it some more. How would you explain what is happening? (You can look at the docs for the JS Math object.)
4. There's actually another place in this code snippet where JavaScript is inserted into the template. Where is it? How would you explain what's happening with this other bit of JS?

Answer on slido.

JSX

As noted, React Native's template system (inherited from React) is JSX. JSX is unusual in how it integrates layout and code. Many other template systems are, in a sense, "template-first" in that templates are defined in files that are first and foremost layout oriented with code sprinkled in. For example, the templates used in JINJA2, are defined in `.html` files. They are primarily HTML files with template syntax embedded, and they are transformed by a pre-processor into full-fledged HTML files. This is how things are done in Angular, too, as well as lots of other modern frameworks. JSX templates, on the other hand, are integrated into JavaScript code files (as you have seen). To do this, JSX uses some pretty neat tricks whereby the template code is translated into React Native objects, which, in turn, are linked to true native UI components.

Any block of valid JSX code can appear in a React Native code file in any place where an object could appear.

Thus, if this is a valid statement:

```
let foo = {a: 1, b: 2};
```

Then this is too:

```
let foo = <Text>a</Text>;
```

So far, we've mainly seen JSX in the return statement for the function `App()`. We can infer (and we'd be right) that this function is called by the React Native framework to render the main UI of the app. Mostly we will see JSX used in this type of situation, but as we build more complicated apps we will want to produce JSX in different parts of the code, but ultimately it all needs to end up being returned from a function that is called by the React Native framework to be of any use.

For now we won't get too much into what constitutes "valid JSX," other than to say that it works pretty much like HTML (the same rules apply regarding tags, closure, attributes, etc.) and that most of the details regarding specific tags and their attributes you will learn over time (and, in many cases, you will have to look up—again and again).

Oh, there is one more very important thing to know about "valid JSX." A valid JSX statement *always* has one and only one top-level parent tag. 父标签只能有一个

This is valid:

```
let hw = <Text>Hello World</Text>;
```

This is valid:

```
let hw = <View><Text>Hello</Text><Text>World</Text></View>;
```

And even this is valid:

```
let hw = <Text>Hello<Text>World</Text></Text>;
```

But is this valid???

```
let hw = <Text>Hello</Text><Text>World</Text>;
```

Nope.

```
```

Announcements

- I assume everyone now has a working expo development environment
  - You'll need it for HW2!
  - If not, get this sorted out right away and don't hesitate to ask for help!
- You might consider installing a simulator (iOS)/emulator (Android)
  - consume a lot of memory and CPU
  - can find instructions online
  - can speed up your development workflow a bit
- All assignments for the rest of the semester are now represented in Canvas
  - Release dates and due dates should be visible, but not content
  - Next week's releases: DYU 3 and Project 1

Break Time

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

<<<<<<<<<<<<<<<<<<<

>>>>>>>>>>>>>>>>>>>> Lovely Layouts
<<<<<<<<<<<<<<<<<<<<<<<<<<<

Shiyu Xu & Shujie Li

Connecting JS logic and UI components

So far we've seen `<View>` and `<Text>` components. These are fine for displaying stuff but they don't really let the user do much. Let's start to look at how React Native handles basic interaction. For illustration purposes we'll use React Native's `<Button>` component, but you'll probably never want to use it again because it's garbage. That is to say, it supports very limited customization and styling, so most developers don't use it. We'll look at an alternative shortly, but for now let's just look at how React Native handles interaction events.

First, you'll need to add `Button` to your imports at the top of App.js.

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

And then...

```
export default function App() {
  return (
    <View style={styles.container}>
      <Button
        title="Press Me!"
        color="blue"
      />
    </View>
  );
}
```

This one doesn't really do anything, but it kinda looks like a button, doesn't it? One fun thing about the `Button` component is that it looks totally different on Android and iOS, and the very few properties it has work totally different on each platform, so for example on iOS the `color` property controls the text color but on Android it controls the background color! And no you can't apply any of your own styles using a `style` property! What fun! What garbage! Anyway, back to business. Now we can add an `onPress` property and assign it to a function that will run when the button is pressed. This should look familiar to you, as it's very similar to browser-based JavaScript. One difference to note is the use of `{}` to surround the function definition—as you'll recall these are needed whenever you want to embed JavaScript inside of JSX.

```
export default function App() {
  return (
    <View style={styles.container}>
      <Button
        title="Press Me!"
        color="blue"
        onPress={()=>console.log("I'm pressed!")}/>
    </View>
  );
}
```

一定要传一个function 而不是function的结果

Previously we saw how you could insert dynamic data by weaving JavaScript into JSX templates using `{}`. Here we are seeing another way that JSX and JS can work together—through callbacks. In this case, we are defining a function (anonymous, arrow) that will be called when the `onPress` event fires—a function that the framework will "call back" into when the time comes. To really unleash the power of callbacks, though, we'll need to start writing our App.js differently and understand how to build and use React Components.

Class Components vs Function Components

In the last lesson you may remember me mentioning offhand that React Native will launch your app using whatever you have designated as the `default export` in your App.js. In fact there are two ways to specify your launch point—using "function components" and "class components." They're both called "components" because everything that can be displayed in React Native is a component—including your app! A `<View>` is a React Native component, as is a `<Text>`, as are a lot of other UI components. When you write

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text
>
      <StatusBar style="auto" />
    </View>
  );
}
```

You are defining a function that returns a `<View>` component, which in turn contains a couple of other components. React Native is able to inspect this component and render it, including all of its contents, onto the screen, applying properties and styles to the components as it goes along.

There is an alternative way to define the Component that will be the launch point of your app, and that's to make the `default export` a **class that extends the `React.Component` class**. The class you define simply needs to provide an 需要继承实现render implementation of the `render()` function (which is one of the methods built into `React.Component`) to provide the visual components (e.g., `<Text>` , `<View>`) that will be displayed. There are several advantages to using a class component over a function component that we will discuss shortly, but first let's look at how it's done:

```
export default class App extends React.Component {
  render() {
    return (
```

```
      <View style={styles.container}>
        <Text>Open up App.js to start working on your app!</Te
xt>
        <StatusBar style="auto" />
      </View>
    );
  }
}
```

Note that we have changed `function` to `class` and added `extends React.Component` to the class declaration. Saying that `App` extends `React.Component` means that `App` is a *subclass* of `React.Component`, which also means that it inherits all of the functionality of `React.Component` (and there's a lot of functionality in there!) and can override the parts that make this `App` special. One of the things that makes this `App` special is how it `render()`s itself—nobody wants to make an `App` that looks like anyone else's `App`, right?

Note also that the return value from `function App()` has simply moved into the `render()` function. We are displaying the same stuff, we're just getting there in a different way.

Change your `function App` to look like the `class App` above, and reload. If you did it right, nothing should have changed.

Component `state`, or why use class components?

So why all the bother? We just added extra complexity to our code and ended up with the same App. What's the point of that?

The short answer is that class components give you more control over your app, in particular they give you more control over how your app's data (or model) is connected to its UI (view) and functionality (controller). Indeed, if you want your app to dynamically update based on what the user does (e.g., text they input, buttons they click, sliders they slide), you are much better off using class components than function components, as the latter don't provide a straightforward way to update the UI on the fly based on changes to the app's data (until fairly recently they didn't

provide a way to do this at all; now there is a way but it's complicated—perhaps another time).

The mechanism that React class components provide for linking app data to the UI is called *state*. Every React Component (including subclasses) contain a special member variable named `state`. `state` is an "dictionary-style" object, and by manipulating `state` in particular ways, e.g., by adding and changing the properties of `state` using the `React.Component` method `setState()`, you can pass information from your App's data model to your App's UI. Moreover, when you manipulate `state` using `setState()`, ReactNative also forces your component to `render()` as soon as possible, meaning that changes to your data model (i.e., changes to `state`) can be reflected in the UI immediately. To see how this works, let's look at an example.

```
export default class App extends React.Component {

  constructor() {
    super();
    this.state = {
      counter: 0
    }
  }

  render() {
    return (
      <View style={styles.container}>
        <Button
          title="+"
          color="red"
          onPress={()=>{
            this.setState(
              {counter: this.state.counter + 1}
            )
```

初始化父类

必须初始化state

state连接了ui和data model，setState可以修改state

```
          }}
        />
        <Text>{this.state.counter}</Text>
      </View>
    );
  }
}
```

Try this out and when you run it click the red + button. Now let's look at what's going on. First, we declare the class, which extends `React.Component`.

```
export default class App extends React.Component {
```

Next we have a constructor. If we are planning to use `state`, we need a constructor, and in that constructor we need to initialize the `state` object with all of the properties we plan to use and set them to their initial values. In this example, we just have one property `counter` which is initialized to `0`.

We also need to call `super()` as the first statement in the constructor so that the `React.Component.constructor()` is called. The superclass constructor does a bunch of stuff to get our Component ready for action, so it's important to remember this call. If you don't call `super()`, your app will break (which will usually remind you to put it in). So here is the constructor, with the call to `super()` and the initialization of `this.state()`.

```
  constructor() {
    super();
    this.state = {
      counter: 0
    }
  }
```

And finally, in the `render()` function, we *use* `state`. First, look at the `<Text>` component that is below the `<Button>` component. The contents of the `<Text>` will be filled in by `{this.state.counter}`. Note the curly braces—remember this tells us that the contents are JavaScript. And since this snippet of code is inside of `render()` which is inside of our `class` definition, `this` will be a reference to the *current object* (i.e., the Component upon which `render()` is being called), and `this.state` will be the very same state we initialized in the constructor.

```
  render() {
    return (
      <View style={styles.container}>
        <Button
          title="+"
          color="red"
          onPress={()=>{
            this.setState(
              {counter: this.state.counter + 1}
            )
          }}
        />
        <Text>{this.state.counter}</Text>
      </View>
    );
  }
```

花括号内为JS代码

Now look at the `onPress` property of the `<Button>`. Note that, first of all, the *value* assigned to `onPress` is a function. Note also that the function has one statement, which is a call to `this.setState()`. The call to `this.setState()` has one argument, which is a *new state object*. `setState()` will use this object to update `state` to *what it should be now*. Additionally, and very importantly, `this.setState()` does one other very important thing, which is that it tells the Component that is referenced by `this` that it should re-render itself, meaning that `render()` will be called (but only *after* the state has been updated).

Thus, when `<Text>{this.state.counter}</Text>` is processed during the execution of `render()` that follows the call to `this.setState()` it will get the new, updated version of `counter`, which is why you see the number on the screen change!

```
  render() {
    return (
      <View style={styles.container}>
        <Button
          title="+"
          color="red"
          onPress={()=>{
            this.setState(
              {counter: this.state.counter + 1}
            )
          }}
        />
        <Text>{this.state.counter}</Text>
      </View>
    );
  }
```

Now You Try #3
Instead of

```
onPress={()=>{
  this.setState(
    {counter: this.state.counter + 1}
  )
}}
```

why can't we simplify to

```
onPress={
  this.setState(
    {counter: this.state.counter + 1}
  )
}
```

期待一个回调函数，但这里直接运行了setState，运行之后还会重新render，进入无限循环

?

In other words, why do we need to wrap `this.setState()` in a function instead of just making `this.setState()` the action associated with `onPress`?

Try replacing the first version with the second and see what happens.

1. Why does the second version not work?

会陷入无限循环

Answer on slido.

To summarize, here are a few things to keep in mind when dealing with `state`:

1. To take advantage `state`, which you will pretty much always want to do, you need to use a class component rather than a function component.*
2. Initialize `this.state` in your Component's constructor. This is the *only* time you should every assign a value to `this.state` directly.
3. Anywhere but the constructor, to assign a value to `this.state`, you *must* call `this.setState()` and pass in an object that represents the *new value*** you want

to assign to `this.state`. This is the *only* way you should ever change the value of `this.state` outside the constructor. 不可以直接修改this.state，除非用react hooks
4. To *read* the value of `this.state`, you can just refer to it directly as `this.state`.

*: technically this isn't entirely true. You can take advantage of `state` inside of function components using "React Hooks," but we're not going to get into those right now.
**: you don't actually have to pass in a complete new state object. You can pass in an object that has only the properties you want to update. Any properties not represented in your new state object will be left as they currently are.

Sidebar

Why do we keep seeing parentheses around blocks of JSX? Strictly speaking they aren't necessary—most of the time. The reason they are usually used for JSX blocks is to avoid problems from *automatic semicolon insertion (ASI),* a largely despised feature of JavaScript that tries to be nice to developers by letting them write syntactically incorrect code (specifically by leaving off semicolons at the ends of lines...sometimes) and have it still work. It's despised because sometimes it would actually be more helpful to get a syntax error telling you to fix your code than to have the code run and do something you didn't expect. Anyway, since JSX is a non-standard part of JavaScript its syntax can confused the JS interpreter, even when JSX-specific processing tools are involved. Enclosing JSX blocks in parentheses eliminates the problems that come from ASI, so it's a recommended practice.

TouchableOpacity: A Better Button?

I mentioned before that the default React Native `Button` is garbage. Sometimes it will do the job, but usually it won't. To create interactive elements that can be styled more flexibly, React Native provides a few options:

- TouchableOpacity
- TouchableHighlight
- TouchableNativeFeedback (Android only)

The differences between these are pretty subtle (and I'm not sure I really understand them), but the idea is more or less the same—each of them acts as a `<View>` that can act as a `<Button>` (in the sense that it responds to `onPress` events and provides

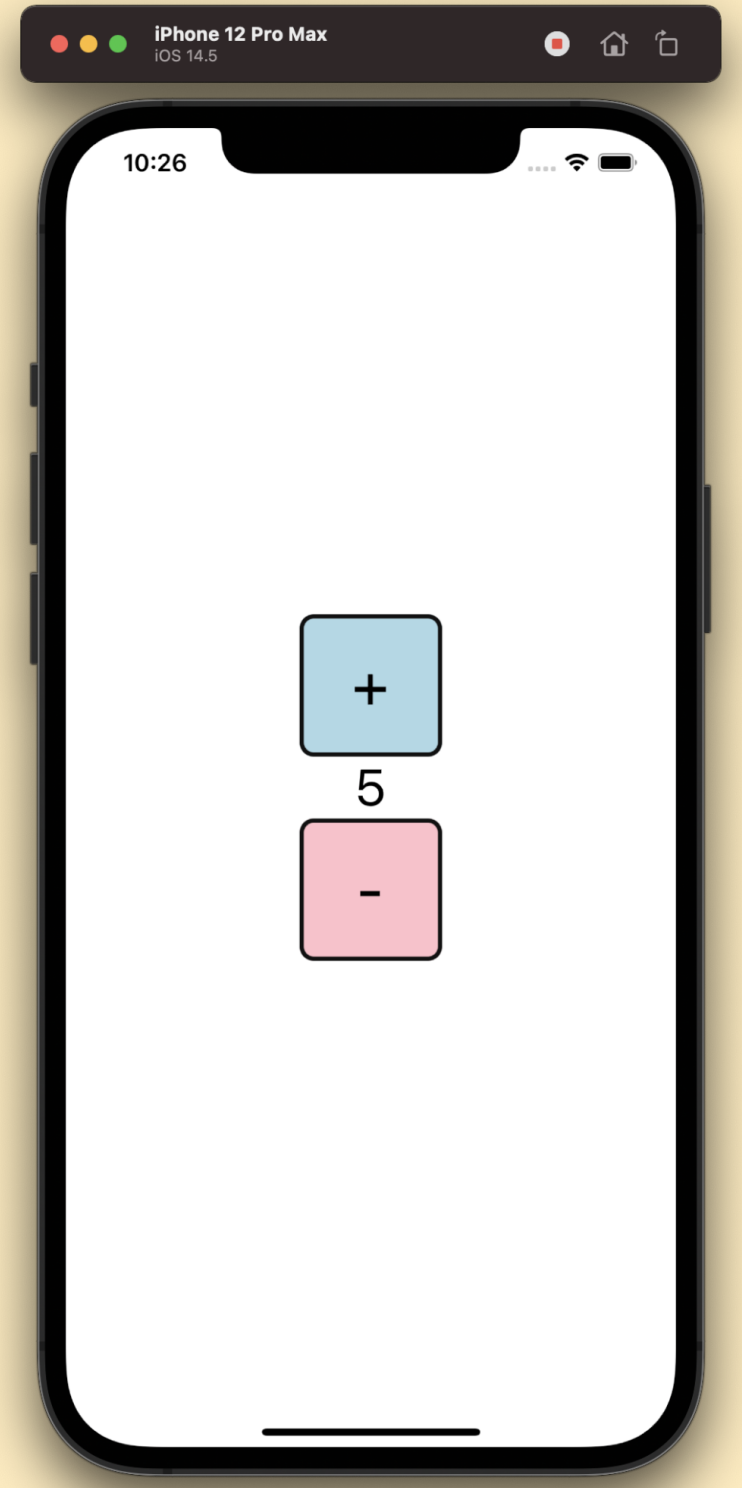
animated feedback to the user to show that a touch was received). Since these elements are essentially `<View>`s, you can put anything in them you want and you can also style them using the full range of style properties available to `<View>`s. We'll focus on `<TouchableOpacity>` for this demonstration. Using a `<TouchableOpacity>`, we can rewrite our counter app like so (using inline styles for readability):

```
  render() {
    return (
      <View style={styles.container}>
        <TouchableOpacity
          style={{
            height: 100,
            width: 200,
            alignItems: 'center',
            justifyContent: 'center',
            padding: 10,
            backgroundColor: 'lightblue',
            borderRadius:10,
            borderWidth: 3,
            borderColor: '#111'
          }}
          onPress={()=>{
            this.setState({counter: this.state.counter + 1}
          )}}
        >
          <Text style={{fontSize: 36}}>{this.state.counter}</T
ext>
          <Text style={{fontSize: 12}}>Press me!</Text>
        </TouchableOpacity>
      </View>
```

```
    );
  }
}
```

Note that there are two `<Text>` elements *inside* the `<TouchableOpacity>`, and that we have a wide range of styling options available for both the `<Text>` and the `<TouchableOpacity>`.

	Now You Try #4	
	Modify the above example to include a “decrement” button as well. Your resulting app should look something like this:	



Unfortunately it won’t work to paste this code into sli.do.

	It seems to reject any text of the form `<tag eventhandler={function}/>`. So just try it and ask any questions that you have if you get stuck.	

You can find a solution to this, as well as all this week's examples at https://github.com/SI669-internal/week4App

Conclusion

Having a solid grasp of JSX, event handling, and state will open a lot of possibilities for building mobile apps with React Native. It's worth reviewing these concepts to make sure you have a solid grasp of these concepts, as nearly everything in React Native builds upon these fundamental concepts. Indeed, we will keep building on these concepts over the upcoming weeks!