Lec 2: Meet, Pull, Hello, Function, State

CPSC 411a, Bruce McKenzie, Fall 2023

Table of Contents

Try to meet me

10 points

- Schedule a meeting with me via Lettuce Meet
- Due:: tonight, 11:59
- Upload the link into the Canvas assignment.
- *Do NOT email it to me, just upload the link*

We will not actually meet at this time. I'm verifying your ability to schedule a meeting

Do a PR

15 points

1. DM me your Github username (in Zoom chat)
    i. Message should be exactly `github bjmckenz` (for example)
    ii. If you're not here synchronously, Canvas it to me
    iii. If you're on the waitlist, email it to me at brucem@fullerton.edu
2. I will add you an team in organization (`23fa-cpsc411`) "right now" (at end of class)

(etc)

Go read the assignment.

- Due Friday, 1159pm

Hello...

- Any problems?
- What are your experiences, feedback?
- How did this match up against "how you expect to learn things"?

My experience

- The app side is easy.
- Setting up `eas.json` was easy, but was it right?
- Spent 2-3 hours on a EPERM (No permission) error in build
    - Turned out to be that I had not (git) committed my code.
    - lots of StackOverflow and Google
    - lots of moving things around on my Windows.
- Another glitch in the `developmentClient` flag
- but then smooth (slow)

Another Perspective

This is about how you can expect to get work assignments. "Go do it" with limited context, and a expectation that you will learn what is necessary.

- It's OK to ask for help
- Expected to ask for "the right amount" of clarifying questions.
- But expected to drive to a terminal point

Functional Programming

Why?

- React/Native is largely based on FP.
- FP is different from what you've ever learned
- If you hack RN without understanding *you will have a bad day*
- FP is useful in many other places
  - Readability, maintainability, testability, reusability
- Downsides
  - Can be snobbish. Math weenies
  - Often VERY terse
  - Steep learning curve
  - JS does not natively enforce FP thinking *(other languages do)*

What?

- Immutability
- Pure Functions
- First-Class & Higher-Order Functions
- Composition

You really ought to dig into Alonzo Church's Lambda Calculus. As important as Turing!

Immutability

- You don't change objects
  - `customer.lastName = "Fred"` *modifies* the object
- Instead, you replace the object
  - `customer = {...customer, lastName: "Fred"}`
  - Yeah, there's a lot to unpack there.
- Why?
  - Anyone with a reference to the old object is safe. It remains consistent, just outdated.

Pure Functions

- Functions do not have *side effects*
- no state, no global variables, no modified parameters
- Why?
  - Easy to parallelize, easy to test.
- ***(Insert Rant about Global Variables)***

OK, You asked for it

- Global variables: Shared non-local state
- Classes are global variables.
- Static vars
- Anything declared in an outer scope
- **The FP weenie will say "This will come back to bite you"**

*True, and **YES,** FP is better, but don't be a snob!*

First-Class & Higher-Order Functions

- First-class functions:
  - A function can be assigned, etc.
  - Not just called
- This means, implies, is for: *you can pass a function into another function to modify the behavior*
  - Or at the very least simplify things

JS Functions How-to

- There are three ways to declare functions.

1. `function add(x,incr) { return x+incr }`
2. `const add = function(x,incr) { return x+incr }`
3. `const add = (x,incr) => x+incr`
    - This is called a **statement function** or a **lambda expression**

These all are basically the same (because of the `const`)

First-Class!

- Did you notice that the second two looked like variable definitions?
- Because *they were!*
- All three could be called as in `y = add(j,45)`
- But they can also be "slung around"
    - `let that_func = add`
    - `y = that_func(j,45)`
    - Same thing as the earlier one!

Higher-Order

- So imagine a **function** that takes (as parameters) **functions** and maybe returns a **function** as a result...
- `const sum_twice(some_func) { return (x,y) => some_func(x,y)+some_func(x,y) }`
- This returns a function that is based on other functions.

Composition

- Chaining results -- that is, passing the results of function 1 into functions 2.
- Mathematically, `f(g(x))`
- JS: `active_cust_addresses = customers.filter( x => x.active ).map( c => cust_address(c) )`
    - See how the array `customers` gets filtered, then the results transformed?
- There's a lot more to this -- this is the "101" version.

State Management (React specific)

- **YOU** don't maintain it. You get things, and set as needed.
- React handles updating it and giving you the most current copy
- Remember earlier: "*You always have a consistent copy*"
- That's true, **if you use and trust React, which you should.**
- ***Don't do it yourself!***

To Do

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

const App = () => {
  const [task, setTask] = useState('');
  const [tasks, setTasks] = useState([]);

  const addTask = () => {
    if (task.trim().length > 0) {
      setTasks([...tasks, { id: Date.now().toString(), value: task }]);
      setTask('');
    }
  };

  const removeTask = (taskId) => {
    setTasks(tasks.filter(t => t.id !== taskId));
  };

  return (
    <View style={styles.screen}>
      <View style={styles.inputContainer}>
        <TextInput
          placeholder="Add a task"
          style={styles.input}
          onChangeText={setTask}
          value={task}
        />
        <Button title="ADD" onPress={addTask} />
      </View>
      <FlatList
        data={tasks}
        renderItem={({ item }) => (
          <View style={styles.listItem}>
            <Text>{item.value}</Text>
            <Button title="X" onPress={() => removeTask(item.id)} />
          </View>
        )}
        keyExtractor={(item) => item.id}
      />
    </View>
  );
};

const styles = StyleSheet.create({
  screen: {
    padding: 50,
  },
  inputContainer: {
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
  },
  input: {
    width: '80%',
    borderColor: 'black',
    borderWidth: 1,
    padding: 10,
  },
  listItem: {
    flexDirection: 'row',
    justifyContent: 'space-between',
    alignItems: 'center',
    borderColor: 'grey',
    borderWidth: 1,
    marginVertical: 10,
    padding: 10,
  },
});

export default App;
```

Header

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

const App = () => {
  const [task, setTask] = useState('');
  const [tasks, setTasks] = useState([]);
```

Whoa, App is a function?

What are those `const` things?

Adding, removing a task

```
const addTask = () => {
    if (task.trim().length > 0) {
      setTasks([...tasks, { id: Date.now().toString(), value: task }]);
      setTask('');
    }
  };

  const removeTask = (taskId) => {
    setTasks(tasks.filter(t => t.id !== taskId));
  };
```

Notice, NO local vars. Just nested functions. We use the state-update functions above!

The Meat: The app returns JSX

```
return (
   <View style={styles.screen}>
     <View style={styles.inputContainer}>
       <TextInput
         placeholder="Add a task"
         style={styles.input}
         onChangeText={setTask}
         value={task}
       />
       <Button title="ADD" onPress={addTask} />
     </View>
     <FlatList
       data={tasks}
       renderItem={({ item }) => (
         <View style={styles.listItem}>
           <Text>{item.value}</Text>
           <Button title="X" onPress={() => removeTask(item.id)} />
         </View>
       )}
       keyExtractor={(item) => item.id}
     />
   </View>
 );
};
```

- Notice use of state variables?
- OK, you got me. `styles` is a module variable.

Making this module an App

```
export default App;
```

- This exposes the App **function** as a symbol. React knows to call it when anything happens.

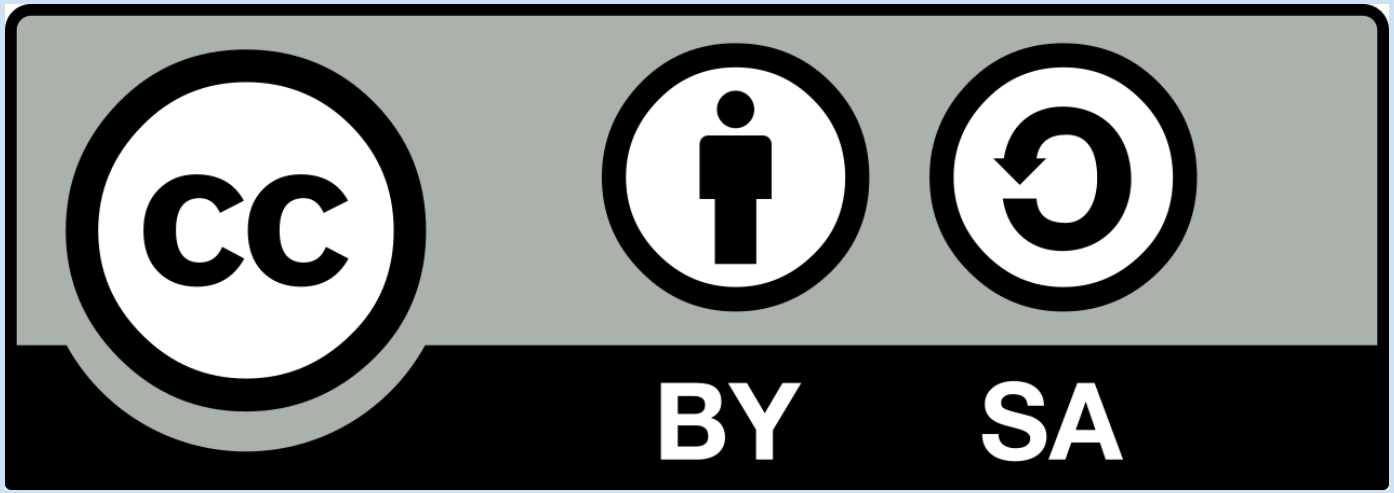
Did you notice?

- Nested functions
- No (local) variables/state
- App func returns "an interface" that can be drawn
- React handles the magic of *reacting* to the user, triggering *just the right thing*

Upcoming

- Multiple screens (navigation)
- more components
- Your informal assignment:: **Get To-do working. Mess around with it. Make it look different. Try it in portrait and landscpe**

License



Author: Bruce McKenzie brucem@fullerton.edu / bjmckenz+pres@gmail.com