CS 4840, Fall 2019 Dr. Zhiguang Xu

Just Enough JSX and ES6

Last Update: Auguest 3, 2019

0. Create a ReactJS app

To create a ReactJS app, you may start by manually installing all the relevant packages and their dependencies as briefly listed in section 2 of the "installation_2" document. Or, you could take advantage of a popular tool called `create-react-app` which automatically pulls in a ton of dependencies such as `babel` , `webpack` , `eslint` , etc.

So,

```
npm install -g create-react-app

create-react-app learn-jsx-es6

cd learn-jsx-es6
```

Thanks to the `create-react-app` tool, Git should've already been initialized. You could run the following command to check:

```
git log
```

Open `./.gitignore` and make sure the following lines are in it:

```
/node_modules
package-lock.json
```

Then,

```
git add -A

git commit -m "created a new react app"

npm start
```

> If for some reason, Git has not been initialized,
>
> 1. Create a new file `./.gitignore` and add the following lines into it:
>
>    ```
>    /node_modules
>    package-lock.json
>    ```
>
> 2. Initialize git by running:
>
>    ```
>    git init
>
>    git add -A
>
>    git commit -m "created a new react app"
>    ```

Start the React Dev server:

```
npm start
```

Now you should be automatically kicked over to the browser that is accessing your React Dev server at `localhost:3000`.

> AWS C9 user: to launch the browser, click "Preview", then "Preview running application".

1. Transpilation

Facebook's React team released JSX when they released React to provide a concise syntax for creating complex DOM trees with attributes. Also, as the JavaScript language evolves, the early adopters in the React community tend to use the new syntax, in particular, the significant ones added in ECMAScript 2015 or ES6 (plus a few added in ECMAScript 2016 or ES7). However, not all browsers support the latest ES6 and ES7 syntax and no browser supports JSX.

Babel is a "transpiler" that

- Converts JSX into pure React elements
- Replaces ES6/ES7 with ES5

such that React apps written with the latest syntax can be executed across all the browsers.

> Although in real world, your app must be tested on ALL browsers before going live online, in this class, we will only use Chrome. Also, **make it habit of displaying the "Developer Tools" whenever you open Chrome to test your code**.

If you are not familiar with the latest JSX and ES6 spec, looking at React code can be daunting. So, in the rest of this tutorial, we will walk through some important and popular JSX and ES6 features. You won't be able to use JavaScript to its best until you understand it's a **functional** language that stresses on the **immutability** of data. We will talk about that too.

So let's get it started with our first taste of JSX.

2. A first and very brief taste of JSX

In conventional Web programming, we are all used to embedding Javascript inside HTML to render dynimic web pages. JSX, on the other hand, allows you to embed HTML inside Javasript.

Open `src/index.js` , the entrance to the "*learn-jsx-es6*" app. The most important line is

```
...
ReactDOM.render(<App />, document.getElementById('root'));
...
```

It finds the element by the id of "root" (in this case, it is an HTML div) of the DOM tree and attach a React component "App" to it.

So, now let's open `src/App.js` . Comment out the entire boilerplate function `App` . Type the following code right above the commented out function. Save the file.

```
const App = () => {
    return <h1>Hi!</h1>
}
```

> Notice that Chrome automatically refreshes every time you make any change to the source code and save it. Error messages, if any, will be displayed in the Developer console. This is called "**Hot Reloading**", thanks to the React Dev server running in the terminal, on which you should Keep your eyes since important info/warning/error messages that you can't afford to miss will also be displayed there.

Resume `src/App.js` back to how it was.

3. Enough JavaScript to become an effective "Full-Stacker"

The three lines of simple code above contains both JSX and ES6 syntax. Go to http://babeljs.io, click "Try it out". Copy the three lines of code above to the left pane and see how Babel tranpiles it into vanilla ES5 JavaScript. By default, Babel has both react (JSX) and es2015 (ES6) enabled. Disable both, enable them back on one by one, check out what you see in the right pane.

We will spend tremendous amount of time on JSX when we start working on a real React project later. For now, let's focus on ES6.

3.1 ES6

Go through **Chpater 2 Emerging JavaScript** in the book "*Learning React, Functional Web Development with React and Redux, by Alex Banks and Eve Porcello*" **thoroughly**.

Here is just a list of new syntax in ES6 that I will highlight and elaborate in class:

- Default parameters
- Arrow functions

  > The discussions about the `this` keyword and lexical scope on pages 16 and 17 are advanced. In Appendix A, please find an in-depth yet brief explanation of this subject exclusively from me.

- Destructing assignment (of objects and arrays)
- The spread operator
- Promises

3.2 What does it mean to be functional/declarative?

Go through **Chpapter 3 Functional Programming with JavaScript** in the book "*Learning React, Functional Web Development with React and Redux*" **thoroughly**.

Here is just a list of new syntax in ES6 that I will highlight and elaborate in class:

- JavaScript functions are first-class citizens
- Imperative (e.g. Java, jQuery) vs. Declarative (e.g. React)
- Immutability
- Pure functions

- Data Transformation with `Array.filter`, `Array.map`, and `Array.reduce`
- Higher-order functions (if time permits)

> **Functional programming is all about transforming data from one form to another while keeping the original intact.** Due to the exceptional importance of data immutability and data transformation in web program, these topics are further discussed in section 3.3 below.

3.3 Why is immutability important during data transformation?

First of all, why is immutability important? Well, mutating data can produce code that's hard to read and error prone. For primitive values (like numbers and strings), it is pretty easy to write 'immutable' code, because primitive values cannot be mutated themselves. Variables containing primitive types always point to the actual value. If you pass it to another variable, the other variable get's a fresh copy of that value, which doesn't have any impact on the original one.

Objects (and arrays) are a different story, they are passed by **reference**. This means that if you would pass an object to another variable, they will both refer to the same object. If you would then mutate the object from either variable, they will both reflect the changes. Example:

```
const person = {
  name: 'John',
  age: 28
};

const newPerson = person;
newPerson.age = 30;
console.log(newPerson === person);    // true
console.log(person);  // { name: 'John', age: 30 }
```

Can you see the problem here? When we change `newPerson`, we also automatically change the old `person` variable. This is because they refer to the same object. In most cases this is unwanted behaviour and bad practice. Let's see how we can solve this.

> Summary: immutable is not equal to read-only.
>
> Although `person` itself is a `const` (i.e. you are not allowed to do something like `person=another;`, in another word, `person` itself is **read-only**), the object that it is referencing is still mutable (i.e. you still can do something like `person.name='Steve'`), which is the exact thing that we want to prevent from happening in React.

3.3.1 Going immutable - objects

3.3.1.1 `Object.assign`

Instead of passing the object and mutating it, we will be better off creating a completely new object:

```
const person = {
  name: 'John',
  age: 28
}
const newPerson = Object.assign({}, person, {
  age: 30
})
console.log(newPerson === person)  // false
console.log(person)  // { name: 'John', age: 28 }
console.log(newPerson)  // { name: 'John', age: 30 }
```

`Object.assign` is an ES6 feature that takes objects as parameters. It will merge all objects you pass it into the first one. You are probably wondering why the first parameter is an empty object `{}`. If the first parameter would be 'person' we would still mutate person. If it would be { age: 30 }, we'd overwrite 30 with 28 again because that would be coming after. **This solution works, we kept `person` intact, we treated it as immutable**!

> A side note: in Javascript, semicolons at the end of statements are optional.

3.3.1.2 Object spread (`...`)

However, EcmaScript actually has a special syntax that enables us to do this even more easily. It's called **object spread**. It looks as follows:

```
const person = {
  name: 'John',
  age: 28
}

const newPerson = {
  ...person,
  age: 30
}
console.log(newPerson === person)  // false
console.log(person)  // { name: 'John', age: 28 }
console.log(newPerson)  // { name: 'John', age: 30 }
```

Again, same result. This time, even cleaner code. First, the *spread* operator (`...`) copies all the properties from `person` to the new object. Then we define a new 'age' property that overrides the old one. Note that order matters, if `age: 30` would be defined above `...person`, it would be overridden by `age: 28` in `person`.

3.3.2 Going immutable - arrays

3.3.2.1 Object spread (`...`)

Let's do a little example of how you could add an item to an array in a mutating way:

```
const characters = [ 'Obi-Wan', 'Vader' ]

const newCharacters = characters

newCharacters.push('Luke')
console.log(characters === newCharacters)   // true :-(
```

The same problem as with objects. We're desperately failing in creating a new array, we just mutated the old one. Gladly ES6 contains a *spread* operator for arrays! Here's how to use it:

```
const characters = [ 'Obi-Wan', 'Vader' ]

const newCharacters = [ ...characters, 'Luke' ]

console.log(characters === newCharacters)  // false
console.log(characters)  // [ 'Obi-Wan', 'Vader' ]
console.log(newCharacters)  // [ 'Obi-Wan', 'Vader', 'Luke' ]
```

Nice, that was easy! We created a new array containing the old characters plus `Luke`, leaving the old array intact.

3.3.2.2 Other operations on arrays

```
const characters = [ 'Obi-Wan', 'Vader', 'Luke' ]

// Removing Vader
const withoutVader = characters.filter(char => char !== 'Vader')
console.log(characters)  // [ 'Obi-Wan', 'Vader', 'Luke' ]
console.log(withoutVader)  // [ 'Obi-Wan', 'Luke' ]

// Changing Vader to Anakin
const backInTime = characters.map(char => char === 'Vader' ? 'Anakin' : char)
console.log(characters)  // [ 'Obi-Wan', 'Vader', 'Luke' ]
console.log(backInTime)  // [ 'Obi-Wan', 'Anakin', 'Luke' ]


// All characters uppercase
const shoutOut = characters.map(char => char.toUpperCase())
console.log(characters)  // [ 'Obi-Wan', 'Vader', 'Luke' ]
console.log(shoutOut)  // [ 'OBI-WAN', 'VADER', 'LUKE' ]

// Merging two character sets
const otherCharacters = [ 'Yoda', 'Finn' ]
const moreCharacters = [ ...characters, ...otherCharacters ]
console.log(characters)  // [ 'Obi-Wan', 'Vader', 'Luke' ]
console.log(otherCharacters)  // [ 'Yoda', 'Finn' ]
console.log(moreCharacters)  // [ 'Obi-Wan', 'Vader', 'Luke', 'Yoda', 'Finn' ]
```

> Make sure you understand the difference between `const moreCharacters = [ ...characters, ...otherCharacters ]` and `const moreCharacters = [ characters, ...otherCharacters ]` and `const moreCharacters = [ characters, otherCharacters ]`
>
> Try them if not.

See how nice these *functional* operators are? The ES6 arrow function syntax makes them even more neat. They return a new array every time you run them, one exception is the ancient **sort** method:

```
const characters = [ 'Obi-Wan', 'Vader', 'Luke' ]
const sortedCharacters = characters.sort()
console.log(sortedCharacters === characters)  // true :-(
console.log(characters)  // [ 'Luke', 'Obi-Wan', 'Vader' ]
```

Yeah, I know. In my opinion `push` and `sort` should have the same behavior as `map`, `filter` and `concat`, return new arrays. But they don't and changing that now would probably break the internet. If you need to use `sort`, you can use `slice` to fix this:

```
const characters = [ 'Obi-Wan', 'Vader', 'Luke' ]
const sortedCharacters = characters.slice().sort()
console.log(sortedCharacters === characters)   // false :-D
console.log(sortedCharacters)  // [ 'Luke', 'Obi-Wan', 'Vader' ]
console.log(characters)   // [ 'Obi-Wan', 'Vader', 'Luke' ]
```

3.3.3 Going immutable - Big advantages in Web apps

One of the complicated operations in developing Web applications is tracking if an object changed. Solutions like `Object.observe(object, callback)` are pretty heavy. However, if you keep your state immutable, you can just rely on `oldObject === newObject` to check if state changed or not, this is way less CPU demanding.

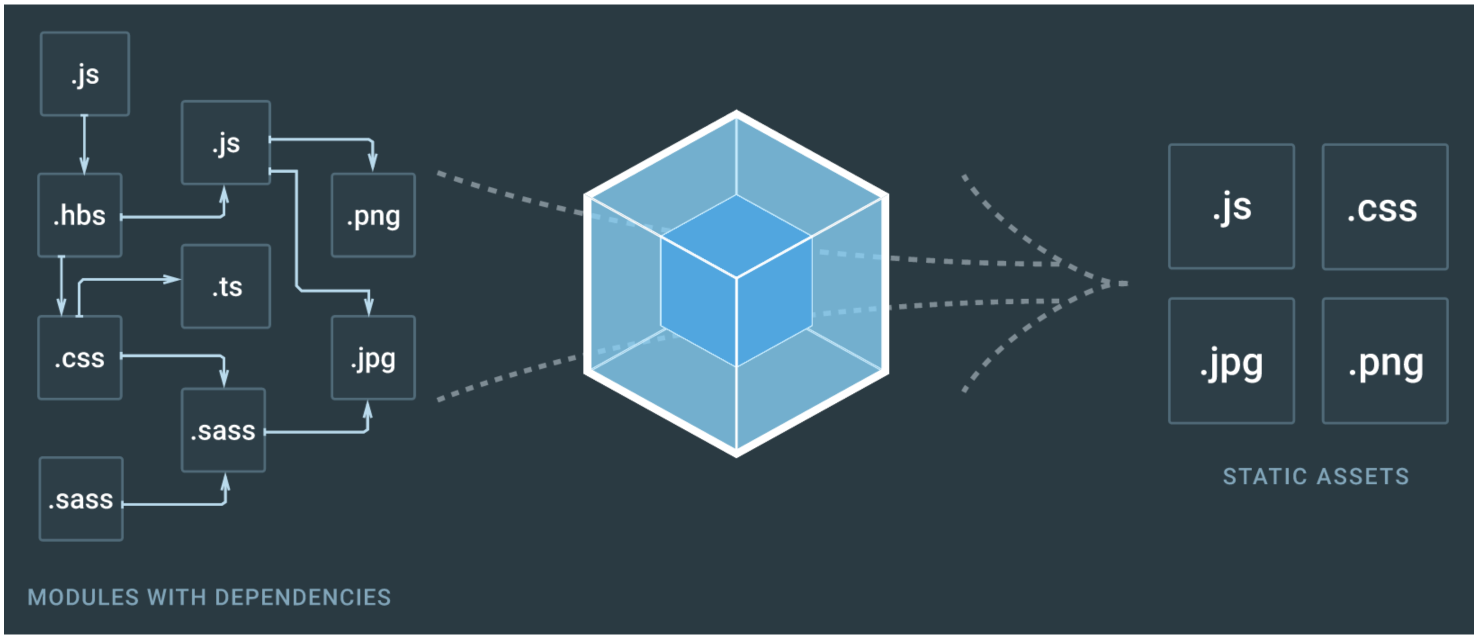
Second big advantage is code quality. Making sure your state is immutable forces you to think better of your application structure. It encourages programming in a more functional way, makes your code easy to follow and reduces the possibility of nasty bugs. Win, win, right?

Okay, to summarize, here is a list of "good" consequently recommended functions as far as data immutability is concerned:

- `Object.assign`
- The spread operator
- `Array.filer`
- `Array.map`
- `Array.slice`
- `Array.reduce`

4. A few words on Webpack

Webpack is a build tool or module bundler that puts *all* the assets in your application, including JavaScript files, images, and CSS, into a dependency graph, spits out one or more bundles (usually one), and serves it to the browser. Why? faster!



The `create-react-app` tool that we use in this class kind of hides the role played by Webpack behind the scenes. We don't even see the configuration file `webpack.config.js` in the project generated by `create-react-app`. We will keep it this way unless some customized adjustments of Webpack turn out to be absolutely necessary.

Appendix

A. The `this` keyword, lexical scope, and arrow functions in ES6

A.1 Direct functions

In the following code, the **normal ES5 function** `print` creates an enclosing lexical context -- the `tahoe` object -- for the `this` keyword inside it. In other word, the lexical scope of `this` is defined as `tahoe` (in plain English, `this` is equal to `tahoe` or in JavaScript `this === tahoe`). Therefore, when being called, the `console.log(...)` statement works.

```
//Will work
let tahoe = {
    resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
    print:  function (delay = 1000) {
        console.log(this.resorts.join(", "));
    }
};

tahoe.print();
```

If we replace the normal ES5 function with an **ES6 arrow function**, the enclosing lexical context is **NOT** defined for `this`. Therefore, when being called, the `console.log(...)` statement will throw an error because you can't refer to `resorts` through an empty `this`.

```
// Won't work
let tahoe = {
    resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
    print: (delay = 1000) => {
        console.log(this.resorts.join(", "));
    }
};

tahoe.print();
```

A.2 Indirect functions

Now, let's make things a little bit more trickier.

In the following code, although the anonymous **ES6 arrow function** (the one passed to `setTimeout` as an argument) does not define an enclosing lexical context for `this`, it does not block the scope of it from propagating up by one level either. So the lexical scope of `this` is defined by the **normal ES5 function** **`print`** -- as you know, the `tahoe` object. Therefore, when being called, the `console.log(...)` statement works.

```
// Will work
let tahoe = {
    resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
    print: function (delay = 1000) {
        // The following line implies window.setTimeout(...
        setTimeout( (delay = 1000) => {
            console.log(this.resorts.join(", "));
        }, delay);
    }
};

tahoe.print();
```

If we use the **ES6 arrow function** syntax at both placed as shown in the following, neither of them will define an enclosing lexical context for `this`. Therefore, when being called, the `console.log(...)` statement will throw an error because you can't refer to `resorts` through an empty `this`.

```
// Won't work
let tahoe = {
    resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
    print: (delay = 1000) => {
        // The following line implies window.setTimeout(...
        setTimeout( (delay = 1000) => {
            console.log(this.resorts.join(", "));
        }, delay);
    }
};

tahoe.print();
```

The following won't work because as stated in the comments, `setTimeout...` actually implies `window.setTimeout...`. Therefore, the enclosing lexical context defined for `this` by the **normal ES5 function** is `window` - a pre-defined object referencing the root of the DOM tree.

```
// Won't work
let tahoe = {
    resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
    print: (delay = 1000) => {
        // The following line implies window.setTimeout(...
        setTimeout( function (delay = 1000) {
            console.log(this.resorts.join(", "));
        }, delay);
    }
};

tahoe.print();
```

Using the **normal ES5 function** syntax at both placed as shown in the following won't work either for the same reason as decsribed above.

```
// Won't work
let tahoe = {
    resorts: ["Kirkwood", "Squaw", "Alpine", "Heavenly", "Northstar"],
    print: function (delay = 1000) {
        // The following line implies window.setTimeout(...
        setTimeout( function (delay = 1000) {
            console.log(this.resorts.join(", "));
        }, delay);
    }
};

tahoe.print();
```

A.3 Conclusion

To prevent the confusion caused by the `this` keyword when working with functions in React, follow the general rules below:

- If `this` is needed in a function that is a direct element in a JavaScript object (as shown in A.1 above), use the **normal ES5 function** syntax.
- If `this` is needed in a function that is indirectly embedded in a JavaScript object (as shown in A.2 above), use the **normal ES5 function** syntax on the first/outermost level and the **ES6 arrow function** syntax on all the other levels.