ECMAScript for React

Credits and copyright

Introduction to React

Flight check

- ➢ Can you see my screen?
- ➢ Can you read my code?
- ➢ Can you hear me?
- ➢ Any distractions?

Schedule

- Morning break
- Lunch break
- Afternoon break

Course preview

- This is the preview for the ECMAScript portion of the class
  - Which runs just today
- Chapter 1: Introduction and setup
- Chapter 2: Modules and scope
- Chapter 3: Classes
- Chapter 4: Functions
- Chapter 5: Objects
- Chapter 6: Array functions

ECMAScript Chapter 1

Introduction and setup

Chapter preview

- Introduction
- What do we want to achieve today?
- Setting up the class
- A little bit about architecture

Introduction

- This is a one-day-long class on modern JavaScript
  - Which we can call ECMAScript 6 … or 7
  - Or 2015, but also with some 2016 & 2017 elements?
  - Or maybe ECMAScript.Next or ECMAScript Harmony
  - We will have to settle the naming system early on
- JavaScript's standard is maintained by the European Computer Manufacturers Association
  - Which used to be abbreviated ECMA, but now styles itself Ecma
  - The name for the language standard for JavaScript is usually written as **ECMAScript**

ECMAScript versioning

- Modern ECMAScript plans to set out a new version of the standard each year
  - ECMAScript 2015, ECMAScript 2016, and so on
- Each year, developers can submit proposals for new JavaScript features
- Those that reach maturity in time, will be part of that year's specification
- The number of specifications for a year can be small, large, or in-between
  - For example: ECMAScript 2016 has only two new features!

Naming schemes

- Some people call what was released as ECMAScript 2015 ECMAScript 6
  - ECMAScript 6 or ES6 is a good shorthand for that large feature set
  - Or ES2015
- Some developers have taken to calling the features specified in ECMAScript 2016 and 2017 as ES7
  - Which is also fine, though we will try to be clear in this course to use ES2016 and ES2017 when clarity is needed or desired

What do we want to achieve today?

- We will mostly be looking at features in ECMAScript 2015
  - Though we will also use either of the features in ES2016
  - And may discuss a few features from ES2017
- The goal is to give you a good grounding in modern JavaScript for use with React
  - You could use React without modern JavaScript, but it is more difficult and less fun
  - As developers, we should plan for the future, and code towards that, rather than looking backwards!

What software will I need?

- **Git**: To check out class files
  - Windows: https://git-scm.com/download/win
  - Mac OS X: https://git-scm.com/download/mac
  - Or `brew install git`
- **NodeJS**: To run the class server
  - Download version 6 or later from https://nodejs.org
  - Make sure that it has been added to your PATH variable
  - Confirm this by opening a terminal window or command prompt and running the following command:

    ```
    node --version
    ```

What software will I need, continued

- **IDE**: There are many JavaScript IDEs, pick one that you like; Here are some that the author of this course uses and likes:
- **WebStorm**: http://jetbrains.com/webstorm
  - Not free
  - Also most other JetBrains products (IntelliJ, for instance)
- **Visual Studio Code**: https://code.visualstudio.com/
  - Free!
  - Based on the editor **Atom** (http://atom.io)
- **SublimeText**: https://www.sublimetext.com/
  - Not free
  - You will have to configure a lot of the IDE yourself

Setting up the class

- The latest version of the class files can be checked out of the GitHub repo:
- **Ask your instructor**
- Check this out to a local directory on your machine
  - On Windows machines, you should check out this repo to the root of the drive you plan to use for class (Probably C:\)
  - This helps avoid a problem with long files paths inherent to Windows
- Mac and Linux machines can check out the files to wherever

Installing dependencies

- Open up a command prompt or terminal window, if you have not already done so
- Change directory to the **sp-es2015** directory
- Enter **`npm install`**
  - If you have **yarn** installed, you can run yarn install instead
- Wait as many, many dependencies for demos and exercises are downloaded
- If there are problems here, check with your instructor; you will need this dependencies, so any issues need to be solved

Windows configuration

- Open up a command prompt
- Change directory to the home directory for the class **sp-es2015**
- Run the command **`bin\set-path`**
- Enter **`webpack --version`** to check that the path has been set correctly (it should report a version number, 2.x.x or later)
- Enter **`start npm run server`**
  - This kicks off the web server in a separate window
- Leave your main terminal/command window open, we will need it later!

Linux and Mac configuration

- Open up a terminal window
- Change directory to the home directory for the class, **sp-es2015**
- Change directory into the **bin** subdirectory
- Ensure you are running **bash**
- Run the command **`source set-path`**
- Enter **`webpack --version`** to check that the path has been set correctly (it should report a version number, 2.x.x or later)

Linux and Mac continued

- **`cd ..`** (back one directory level)
- **`npm run server > http.log 2>&1 &`**
- Runs the **`server`** task in the **package.json** **`scripts`** section, sending both standard output and error output to **http.log**
- Also runs the task in the background (the ending ampersand)
- Leave your main terminal/command window open, we will need it later!

Running a demo

- In your terminal window / command prompt
- **`cd`** into the **`demos`** directory
- Run **`webpack src/installed.js`**
- Open a web browser (Chrome is preferred)
- Navigate to **http://localhost:8000/demos**
- You should see page indicating that the web server is running successfully
  - If you see something that looks like a demo, but does not say that the class files have loaded successfully, mention this to your instructor!

A little bit about architecture

- Our class will use a suite of tools
- **Node.js** plus **http-server**
  - Node.js is a JavaScript runtime, akin in some ways to a virtual machine
  - http-server is a lightweight web server that runs on top of Node.js
- **Babel**
  - Babel is a **transpiler** which can take ECMAScript 2015+ code and **transpile** it to code which an ECMAScript 5 browser can run
  - Babel requires Node.js to run

Even more architecture

- **Webpack** is a tool which packages several layers of JavaScript, CSS, and HTML together to make **easily-deployable applications**
- We will use Webpack to
  - Transpile our ECMAScript 2015+ code to ECMAScript 5-compatible code
  - Package this code into a file which we can view in a browser
- **ESLint**, which helps to enforce good code practice
- These are part of our toolset, they are not the focus of the course

Cheat sheet

- To run any demo:
  - In the **demos** directory, run **`webpack src/name-of-demo`**
  - Then navigate to **http://localhost:8000/demos**
  - If you want to make changes to the demo you can run **`webpack --watch src/name-of-demo`**, which will watch the file for changes
- To run any exercise:
  - In the exercises directory, run **`webpack --watch ex-##/ex-##-index`** where ## represents the exercise number you are on
  - Then navigate to **http://localhost:8000/exercises**

Conclusion

ECMAScript Chapter 2

Modules and scope

Chapter preview

- Problems with scope
- Improving scope with **`let`** and **`const`**
- Problems with libraries
- Building a module
- Using modules

Problems with scope

- Before ES 2015 JavaScript had a big problem with scope
- There were only two scopes: global, and functional
- **No block scope**!
- This often lead to problems with leaking scopes, or scopes that did not act in the way programmers intended or anticipated

Improving scope with let and const

- ES 2015 adds two new scoping commands: **`let`** and **`const`**
  - **`var`** still works, but should be avoided
- **`let`** scopes variables the way you would expect them to be scoped: to the nearest enclosing block
- **`const`** works as **`let`** does, with the added feature of identifying a variable whose value will not change
  - **`const`** permits JavaScript interpreters to optimize variable access
  - **Primitives** (strings, numbers, booleans) cannot change
  - **Object references** (objects, arrays, Dates, etc.) cannot change their reference, but can change their state
  - Where possible, prefer **`const`**, then **`let`**

Demo: Variable scoping

- File: **demos/src/variable-scoping.js**
- From the demos folder, run
  `webpack src/variable-scoping.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Problems with libraries

- Because of the way scoping worked in JavaScript, before ES6, libraries posed many problems
- Libraries had to be global, to expose functionality
- But global libraries raised the problems common to any global variables

Building a module

- ECMAScript 2015 adds **modules** to JavaScript
- Defining a module is simple: write JavaScript code as you would normally
- The code in the module should be scoped with let or const as normal
- When you want to make code from this module available, use the **`export`** command
  - Usually at the bottom of the module file
- Anything you export can be re-used in any code which depends on this module

export syntax

- The statements below explicitly export identifiers from a module
- **`export { name1, name2, name3, ... }`**
- **`export { var1 as name1, var2 as name2, ...}`**
- **`export let name1, name2, name3, ...;`**
- **`export let name1 = 'foo', name2 = 'bar', ...`**

Using modules

- Using modules is quite simple as well
- Use the **`import`** command to import the content you need from the module you wish to use
- Depending on the module's export syntax, you may import the items you need in slightly different ways
- For all of the exports on the previous slide, the following will work:
- **`import {name1, name2} from './other-slide'`**
- You do not need to import all exports

import syntax

- **`import {name1, name2} from './foo';`**
- **`import {name1 as bar} from './foo';`**
  - **`name1`** is aliased to **`bar`**
- **`import * as bar from './foo';`**
  - Items from **`foo`** are now available as **`bar.baz`**
- **`import './foo';`**
  - Runs the code in **`foo`**, imports no actual members

File relationships

- Importing can be done relative to the current file
  - **`import {abc} from './utils';`**
  - Assumes **`utils.js`** is in the same folder as the importing file
- Or importing can be done relative to a base location
  - **`import {abc} from 'utils';`**
- With webpack, the default search location is the project's node_modules folder, where all installed libraries are
  - It is possible to change this configuration if needed
  - Consult webpack's documentation

Exporting and importing defaults

- You can set one and only one standard export for module
  - **`export default <expression>`**
  - **`export default function() { ... }`**
  - **`export default function name() { ... }`**
  - **`export { name1 as default }`**
  - **`export default name1`**
    - The two points above assume that **`name1`** was already defined
- Importing defaults is slightly different
  - **`import defaultThing from 'foo';`**
- Though there is only one default export, there can be other, non-default exports

More importing

- **`import defaultThing, {other1, other2} from 'foo';`**
  - Import the **`defaultThing`**, as well as two other exports (which are not the default)
- **`import defaultThing, * as bar from 'foo';`**
  - **`defaultThing`** is available, all other exports are available as **`bar.whatever`**

Demo: Importing and exporting

- Files:
  - **demos/src/import-export.js**
  - **demos/src/export-no-default.js**
  - **demos/src/export-with-default.js**
- From the demos folder, run
  `webpack src/import-export.js`
  - Note that **webpack** will pick up all the other files
- Then navigate to **http://localhost:8000/demos** to see the results

Export best practices

- If your file is defining a class, export that class as a default
  - You should not define more than one exportable class for a file
- If your file is only exporting one item, export that one item as a default
- If your file is exporting many items, strongly consider having no default
- Note: Sometimes we will violate this best practice in class to make a demo or exercise a bit simpler

Exercise 1: Scope and modules

- File location: **exercises/ex-01**
- Start in **exercises/ex-01/ex-01-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    webpack --watch ex-01/ex-01-index
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Conclusion

ECMAScript Chapter 3

Classes

Chapter preview

- “Classes” in older JavaScript
- Classes in ECMAScript 2015
- Constructors
- Getters and setters
- Inheritance

“Classes” in older JavaScript

- Before ECMAScript 2015, JavaScript did not have classes
- JavaScript did have types, which could be created from functions
- And objects could be used as the basis for a type as well
- It was somewhat confusing
- As Douglas Crockford has said: “JavaScript is an object-oriented language conflicted about its prototypal nature”
- ECMAScript 2015 improves things considerably

Classes in ECMAScript 2015

- With ECMAScript 2015, JavaScript adds proper classes
- Well, class syntax, the actual underlying implementation still uses JavaScript's prototypal nature to get things done
- The new syntax is a great improvement over having to jump through weird syntactical hoops to implement classes

Defining a class

- `class Foo { /* class definition here */ }`
- That's it!
- Use **`export`** to export it as a member of a module
- Use **`export default`** to export it as the default member of a module
  - Define only one exportable class per module
- Define methods inside the class
-
```
class Foo {
  bar() { /* method definition here */ }
}
```

Constructors

- Classes usually use constructors to create instances
- JavaScript classes reserve the **`constructor()`** function to create instances
  - If a constructor is not defined, a default constructor is provided
- Within the constructor, the **`this`** keyword refers to the instance the constructor is creating
- There is no need for a **`return`** statement
  - **`return this;`** is implicit

Constructor best practices

- How many arguments should my constructor have?
- If your class can be instantiated with **three or fewer** arguments, then use those arguments in the constructor
- If your class requires four or more arguments to be instantiated, use a configuration object
- Pass in an object literal to the constructor, like so

  -
    ```
    let honda = new Car({
    make: 'Honda',
    model: 'Civic'});
    ```
- In the constructor function, copy the values from the object literal to the class instance

Demo: Defining a class

- Files:
  - **demos/src/define-class.js**
  - **demos/src/Car.js**
- From the demos folder, run
  `webpack src/define-class.js`
  - Note that **webpack** will pick up all the other files
- Then navigate to **http://localhost:8000/demos** to see the results

Getters and setters

- JavaScript does not have visibility modifiers
  - Thus, it is difficult to have private members of a JavaScript class
- The special methods **`get()`** and **`set()`** control accessors for properties within a class definition
  - **`get foo() { /* return something here */ }`**
  - **`set foo(val) { /* set foo here */ }`**
- These getters and setters will be used on property access
  - **`obj.foo`** calls the **`get()`** method above
  - **`obj.foo = 'bar'`** calls the **`set()`** method above
- But watch out for recursion!

Rules for getters and setters

- Getters take no parameters
- Setters take exactly one parameter
- Both can be deleted with **`delete`** (though not separately)
- Use **`Object.defineProperty()`** to add a new property with a getter and/or a setter to an existing object
- You cannot define a getter or setter for a property that otherwise has a value
- Do not add a getter or setter which calls itself:

```
get make() {
  return this.make // Infinite recursion!
}
```

Demo: Getters and setters

- Files:
  - **demos/src/getters-setters.js**
  - **demos/src/Car-with-getters-and-setters.js**
- From the demos folder, run
  `webpack src/getters-setters.js`
  - Note that **webpack** will pick up all the other files
- Then navigate to **http://localhost:8000/demos** to see the results

Exercise 2: Building a class

- File location: **exercises/ex-02**
- Start in **exercises/ex-02/ex-02-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    webpack --watch ex-02/ex-02-index
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Inheritance

- Before ECMAScript 2015, inheritance was particularly thorny
- Now, inheritance is relatively simple
- If your class inherits from another class, it **`extends`** the superclass
  - **`class Car extends Vehicle { ... }`**
- In the subclass's constructor, you can invoke the superclass's constructor via **`super()`**
- In a subclass method, you can invoke the superclass's method by calling **`super.methodName()`**
- JavaScript only supports **single inheritance**

Demo: Inheritance

- Files:
  - **demos/src/inheritance.js**
  - **demos/src/Car-inheritance.js**
  - **demos/src/Vehicle-inheritance.js**
- From the demos folder, run
  `webpack src/inheritance.js`
  - Note that **webpack** will pick up all the other files
- Then navigate to **http://localhost:8000/demos** to see the results

Exercise 3: Inheritance

- File location: **exercises/ex-03**
- Start in **exercises/ex-03/ex-03-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    webpack --watch ex-03/ex-03-index
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Static methods

- Static methods can be added to a class by labeling them with the keyword **`static`**
- Static methods belong to the class, not to any instance
- Static members cannot be called from an instance, only from the class
- Use static methods to define class-wide utilities

Static fields

- What about static fields (or properties)?
- ES 2015 does not have a proposal for static fields
- But you can get the same effect two ways:
- Define a property on the class instance itself
  - **`class Foo { … }`**
  - **`Foo.staticProperty = '…'`**
  - **`Object.defineProperty(Foo, { … })`**
- Define a **`static`** getter and/or setter for the static property

Conclusion

ECMAScript Chapter 4

Functions

Chapter preview

- The spread operator
- Rest parameters
- Default values for parameters
- Arrow functions (sometimes “fat arrow” functions)
  - Why arrow functions?
  - Defining arrow functions
  - Use cases

Before the spread operator

- Before ECMAScript 2015, it was complicated to add one array to another
  - **`let newArray = arrayOne.concat(arrayTwo)`**
- Not to mention adding multiple elements with **`push()`**
  - **`numbers.push([1, 2, 3])`**
  - The above adds a single element, **which is an array reference**, to **`numbers`**
  - **`Array.prototype.push.apply(numbers, [1, 2, 3])`**
  - The above code invoked **`push()`** with the array spread out as individual arguments

The spread operator

- The spread operator makes using multiple arguments easy
- **`numbers.push(...[1, 2, 3])`**
- **`anyFunction(...setOfArguments)`**
- Adding two arrays together
  - **`let newArray = [1, 2, 3, ...otherNumbers, 4, 5, 6];`**
- Cautions:
  - The spread operator works **shallowly**, beware deep structures!
  - The spread operator only works on **iterables** (Array, TypedArray, String, Map, Set)

Demo: The spread operator

- File: **demos/src/spread-operator.js**
- From the demos folder, run
  `webpack src/spread-operator.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Rest parameters

- JavaScript does not enforce method signatures
  - Argument names are a convenience, not a contract
- Rest parameters allow us to specify one parameter for any and all remaining arguments
  - **`function foo(a, b, ...c)`**
  - **`foo('one', 'two', 'three', 'four', 'five')`**
  - Within **`foo()`**, **`c`** is a **three element long array** containing **`'three'`**, **`'four'`**, and **`'five'`**
  - Arrays created from rest parameters have array functions, unlike the arguments object

Demo: Rest parameters

- File: **demos/src/rest-parameters.js**
- From the demos folder, run
  `webpack src/rest-parameters.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Default parameter values

- In function definitions, you can also set default parameter values
- `function(x = 1, y=2) { ... }`
- If the function is invoked with an argument, that argument supersedes the default
- Otherwise, the default value is used

Exercise 4: Spread and rest

- File location: **exercises/ex-04**
- Start in **exercises/ex-04/ex-04-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    webpack --watch ex-04/ex-04-index
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Arrow functions

- ECMAScript 2015 introduced a new function definition syntax
-
```
(arguments) => { /*
  function definition here
*/ }
```
  - There are shortcuts and variations which will be addressed in a later slide
- For reasons we will discuss soon, this definition works well for standalone functions
- It does not work well for defining methods on an object or a class

Why use arrow functions?

- More concise syntax, saving writing the word “function” many times in your code
- Does not bind its own **`this`** object, avoiding a complex, subtle bug
  - Normally, functions bind their own **`this`** object
  - Which means that for functions, inside methods, inside objects, the **`this`** keyword points to the method, not the object
  - The previous solution to this was to use a closure
  - Now obviated by arrow functions

Defining arrow functions

- Basic form:
  `(arguments) => { /* function definition */ }`
- **`(arguments) => expression`**
  - **`expression`** is implicitly prefixed with **`return`**
- **`oneAgument => { /* definition */ }`**
  - Skip the parens when there is only one argument
- **`oneArgument => expression`**
  - Combine a single argument (no parens) with a return expression (no curly braces, no “return” keyword)

Defining arrow functions, continued

- **`() => { /* definition */ }`**
  - If there are no arguments, the parentheses are required
- **`(arguments) => ({ foo: bar })`**
  - Implicitly return an object
- **`let foo = (args) => { /* definition */ }`**
  - Assign an arrow function to a variable
  - Can now invoke **`foo()`**
- **`let bar = oneArg => expression`**
  - One argument (no parens)
  - return expression (no curly braces, no “return” keyword)
  - Can invoke **`bar()`**

Use cases

- There are many use cases for arrow functions, but here are two
- Promises

  -
    ```
    fetch().then((response) => {
      /* do something with response */
    })
    ```
- Array manipulators
  - We will be seeing more about array functions later
  - For now: **`numbers.map(num => doSomething);`**

Demo: Arrow functions

- File: **demos/src/arrow-functions.js**
- From the demos folder, run
  `webpack src/arrow-functions.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Exercise 5: Arrow functions

- File location: **exercises/ex-05**
- Start in **exercises/ex-05/ex-05-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    npx webpack --watch ex-05/ex-05-index.js
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Conclusion

ECMAScript Chapter 5

Objects

Chapter preview

- Copying and merging objects
- Controlling access to objects
- Object destructuring

Copying objects

- ECMAScript 2015 introduced a new static method to copy and merge objects: **`Object.assign()`**
- **`Object.assign(target, ...sources)`**
  - **`sources`** is one or more other objects
- The **enumerable own properties** from **`sources`** will be copied over to **`target`**
  - If two or more sources have the same property, the latest property will overwrite all previous version
- Provide an empty object **`{}`** as a target to shallowly clone a source

Object spread operator

- Objects can also use a spread operator, similar to the way that the array spread operator works
- **`let clone = { …obj }`**
- The variable **`clone`** is now a cloned version of **`obj`**
- This is an idiomatic way of copying an object
- As with the array spread operator, this is a **shallow copy**

Controlling access to objects

- There are three levels of control over an object in JavaScript
  - No new properties
  - No reconfiguration of current properties
  - No modification of property values
- Correspondingly, there are three methods to limit access to an object
  - **`Object.preventExtensions()`**
  - **`Object.seal()`**
  - **`Object.freeze()`**

Object.preventExtensions()

- Use **`Object.preventExtensions(target)`** to prevent adding new properties to an Object
- Existing properties can still be modified
- Existing properties can be deleted
- Attempting to add new properties will silently fail, or throw an exception if in strict mode
- Detect if an Object is extensible via **`Object.isExtensible(target)`**

Object.seal()

- **`Object.seal`** does everything that **`Object.preventExtensions`** does and more
- **`Object.seal`** prevents existing properties from being reconfigured
- Not only can you no longer add properties, the nature of existing properties as configured via **`Object.defineProperty`** / **`Object.defineProperties`** can no longer be changed
- That said, the value of properties on a sealed object can still be changed, as long as said properties were configured as writable
- Detect if an Object is sealed via **`Object.isSealed(target)`**

Object.freeze()

- **`Object.freeze()`** does everything that **`Object.seal()`** does and more
- Frozen objects cannot be modified in any way, making them effectively immutable
  - This is a shallow freeze, which does not affect objects within objects
- Attempting to change the Object will fail silently, or throw an error under **`use strict`**
- Detect a frozen Object with **`Object.isFrozen(target)`**

Demo: Object access

- Files:
  - **demos/src/object-access.js**
  - **demos/src/car-factory.js**
- From the demos folder, run
  ```
  webpack src/object-access.js
  ```
  - Note that **webpack** will pick up all the other files
- Then navigate to **http://localhost:8000/demos** to see the results

Object destructuring

- ECMAScript 2015 introduced new syntaxes for accessing objects and their properties, collectively known as **object destructuring**
- In general, destructuring helps to unwind objects, swap variables, and assign values without temp variables
- The next few slides will show some use cases for object destructuring

Object destructuring, continued

- Unwinding an object

```
const person = {
  firstName: 'Jane',
  lastName: 'Doe'
};
const { firstName, lastName } = person;
// The variables firstName and lastName now exist
// outside of person

const { firstName: givenName, lastName: surname } = person;
// The variables givenName and surname now exist
// outside of person
```

More object destructuring

```
let args = { match   : 'foo',
             history : 'bar',
             location: 'baz' };
someFunction( args );

function someFunction( { match, history } ) {
  // The variables match and history
  // are available here
  // The function does not care about location
}
```

Array destructuring

- Arrays can be used to flip values

```
 x = 10, y = 20;
 [ y, x ] = [ x, y ];
// y === 10 && x === 20
```

- Slice and dice arrays

```
let letters = [ 'a', 'b', 'c', 'd', 'e' ];
let [first, , third] = letters;
// first === 'a' && third === 'c'
```

Demo: Destructuring

- File: **demos/src/destructuring.js**
- From the demos folder, run
  `webpack src/destructuring.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Conclusion

ECMAScript Chapter 6

Arrays

Chapter preview

- Basic array rules
- Generic access: splice and slice
- Manipulating: join, concat, reverse, sort,
- Finding: indexOf, lastIndexOf
- Accessing at the ends: push, pop, shift, unshift
- Iterating: forEach, every, some
- Processing: map, filter, reduce, reduceRight

Generic array access

- **`array.slice(begin, [end])`**: Extract a slice of an array
  - If **`end`** is omitted, returns from **`begin`** to the actual end of the array
- **`array.splice(begin, length, [item, item, item,....])`**: All-purpose array modifier
  - Start at **`begin`**, go **`length`**, possibly inserting/replacing with **`item`**…
  - Insert: **`array.splice(begin, 0, item...)`**
  - Update: **`array.splice(begin, 3, item1, item2, item3)`**
  - Delete: **`array.splice(begin, 2)`** Note that this is the only way to delete elements in an array before ECMAScript 2015
    - Returns the deleted elements as an array in this case

Manipulating arrays

- **`array.join(interString)`**: Joins all elements on **`interString`** and returns a single string
- **`originalArray.concat(otherArray)`**: returns an array consisting of **`originalArray`** with **`otherArray`** concatenated to it
- With ECMAScript 2015 and the spread operator, this gets easier
  - **`let newArray = […originalArray, …otherArray];`**
- **`array.reverse()`**: Reverses the order of elements of an array <u>in place</u>

Sorting arrays

- **`array.sort([compareFn])`**: Sorts the elements of an array in place
  - If **`compareFn`** is not provided, elements are converted to strings and compared by position on the Unicode table
- **`compareFn(a,b)`** can return the following:
  - If **`a`** should be sorted before **`b`**, then a number greater than zero
  - If **`a`** and **`b`** should stay in the same position, relative to one another, then zero
  - If **`a`** should be sorted after **`b`**, then a number less than zero

Demo: Accessing and sorting arrays

- File: **demos/src/array-access-sort.js**
- From the demos folder, run
  `webpack src/array-access-sort.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Searching arrays

- Search through arrays with **`indexOf`**
- **`array.indexOf(element, [start])`**: Returns the position of element if it is found in the array, **`-1`** otherwise
  - Provide a starting index if you would prefer to start at an index greater than zero
  - Searches left to right (index 0 and greater) normally
  - Equality is determined by triple equals
- **`array.lastIndexOf(element, [start])`**: As indexOf, but searches right-to-left (last index and descending)
  - Still returns **`-1`** if the element is not found

Searching arrays, continued

- In ECMAScript 2016 (not a typo!), we also have **`Array.prototype.includes()`**
- Where **`indexOf()`** returns the position of the element in the array, **`includes()`** simply returns true or false based on the presence of the element in the array
- Using **`includes()`** simplifies **`if`** statements when searching for an element in an array
- The **`includes()`** method can also detect **`undefined`** and **`NaN`** in an array, where **`indexOf()`** cannot

Arrays as stacks

- JavaScript arrays have the traditional stack manipulation functions
- **`array.push(element)`**: Adds **`element`** to the end of **`array`**
- **`array.pop()`**: Removes and returns the last element of **`array`**
- **`array.unshift(element)`**: Adds **`element`** to the beginning of the **`array`**
- **`array.shift()`**: Removes and returns the first element of the array, all other elements shift down one index

Demo: Finding elements

- File: **demos/src/array-finding.js**
- From the demos folder, run
  `webpack src/array-finding.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Exercise 6: Using arrays

- File location: **exercises/ex-06**
- Start in **exercises/ex-06/ex-06-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    npx webpack --watch ex-06/ex-06-index.js
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Functional iteration

- You can iterate over an array with a simple `for` loop
- JavaScript arrays also have functional iterators, which are popular
- `array.forEach( (item, index, array) => { ... }, [scope])`: Iterate over `array`, calling `function` once for each element
  - Function has three arguments, the `item`, its `index` in the array, and a reference to the original `array`
  - Provide an optional `scope` argument to have the function operate in other than the current scope / context
  - No return value
  - No way to break out of the iteration

Processing arrays with map

- **`array.map( (item, index, array) => { ... }, [scope])`**: Iterates over the array, building a new array consisting of what **`function`** returns for each **`item`**
- Put more simply, calls the function against each **`item`** and collects the return values into a new, returned array
- While this generates a new array, objects retain their references
- As with **`forEach`**, there is no way to break out of the iteration over the array

Functional iteration with exits

- The **`array.some`** and **`array.every`** functional iterators take the same arguments as **`array.forEach`**
- But they can break if their processing function returns a value
- **`array.some`** breaks its loop if the processing function returns truthy, and itself returns **`true`**
- **`array.every`** breaks its loop if the processing function returns falsy and itself returns **`false`**

Finding an element in an array

- ECMAScript 2015 introduced the **`find`** array method
- The **`find`** method takes a predicate function that should return true or false
- Once the function returns **`true`**, find quits and returns the array element that it was on at the time
- Useful for finding the first element which matches a condition
- The **`find`** method is not natively supported by Internet Explorer, so consider using **`some`** or **`every`** (both supported since version 9) if you have to support IE

Filtering arrays

- **`array.filter( (item, index, array) => { ... }, [scope])`**: Returns a new array of filtered elements based on whether the processing function returns true (filtered <u>in</u>) or false (filtered <u>out</u>)
- The processing function can return truthy or falsy
- There is no way to break out of the loop once it has started, much like with **`forEach`**
- Use find to **`find`** one element that matches a condition, **`filter`** to find **all** elements which match a condition

Reducing arrays

- **`array.reduce( reducerFn, [initialValue])`**
- Iterates over array, calling **`reducerFn`** on each element, building an accumulating value for each element, until it returns the final accumulation of all elements
- The **`reducerFn`** receives four arguments:
  - **`previousValue`**: The previously accumulated value
  - **`currentValue`**: The current element being examined
  - **`index`**: Position of **`currentValue`**
  - **`array`**: Reference back to the original array
- You can use **`initialValue`** to seed the first call of the reducer function
- **`reduceRight`** is the same, except it processes elements right-to-left

Demo: Processing arrays

- File: **demos/src/array-processing.js**
- From the demos folder, run
  `npx webpack src/array-processing.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Exercise 7: Processing arrays

- File location: **exercises/ex-07**
- Start in **exercises/ex-07/ex-07-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    npx webpack --watch ex-07/ex-07-index.js
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Conclusion

ECMAScript Chapter 7

fetch() and Promises

Chapter preview

- Asynchronous JavaScript
- Introducing fetch()
- Using fetch() currently
- Promises
- Chaining promises

Asynchronous JavaScript

- Using JavaScript to asynchronously access data on the server has been a common feature of client-side applications for years
  - Asynchronicity as a feature has spread to other aspects of JavaScript, particularly the UI
- Most refer to this feature as AJAX: **Asynchronous JavaScript and XML**
- Ajax was implemented through a Microsoft-proposed feature: the **`XMLHttpRequest`** object
  - This was originally an ActiveX object that was eventually released and standardized by the World Wide Web Consortium (W3C)

Aging Ajax

- Ajax as implemented through **`XMLHttpRequest`** has not aged well
- The API is clunky, event-based, and not in-line with current idiomatic JavaScript
- Recently, Ajax was upgraded to “Level 2” or version 2
- While this introduced needed new features, it also exposed the limits of working through the **`XMLHttpRequest`** object

Introducing fetch()

- The **`fetch()`** method, available on the **`window`** and the **`GlobalFetch`** objects, is a drop-in replacement for **`XMLHttpRequest`**
- It is more extensible and future-facing than **`XMLHttpRequest`**
- It is also more efficient, covering typical use cases, now that Ajax patterns are widely known and understood
- While there is a specific **`fetch()`** method, the API as a whole is often referred to as the Fetch API

Fetch API status

- The Fetch API is not implemented across all browsers:
  - Chrome 42+
  - Firefox 39+
  - IE Edge 14+
  - Not old IE
  - Not Safari (iOS or MacOS)
  - Android Browser 53+
  - Chrome for Android 53+

The Fetch polyfill

- We will use a polyfill library for fetch called … fetch!
- https://github.com/github/fetch
- We can import it directly as **`fetch()`**, so that when browsers widely support the Fetch API, we can simply drop the import
- The polyfill API completely replaces and mimics the real API

Using fetch()

- **`fetch(url, [config]).then(onSuccess, onFailure)`**
- Call **`fetch()`** passing it a URL and an optional configuration
- **`fetch()`** returns a Promise (more on these soon), which includes a **`then()`** method
- The **`then()`** method takes two arguments, a handler for success and a handler for failure
  - Neither is required
  - **`onSuccess`** (the promise **resolved**) is passed a **`Response`** object
  - **`onFailure`** (the promise is **rejected**) is passed a **`TypeError`** error

A full fetch() example

-
  ```
  fetch( url ).then( ( response ) => {
    return response.json();
  }, ( error ) => {
    console.log( 'Error: ', error );
  });
  ```
- The **`response`** object includes methods for accessing the data of the response
- The error will be a **`TypeError`**, and will only occur if there is some sort of network error, **not** if there is a >= 400 response status

Demo: fetch() in action

- File: **demos/src/fetch-demo.js**
- From the demos folder, run
  `npx webpack src/fetch-demo.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Handling success

- **`onSuccess( Response ) { ... }`**
- The Response object has the following methods
  - **`Response.clone()`**: Clones this response, allows you to re-use the response data (see the next slide), which is usually one-time-use only.
  - The other two response methods, below, are more relevant for the **`ServiceWorker`** API, which we are not covering
  - **`Response.error()`**: Creates a new Response with a network error
  - **`Response.redirect()`**: Creates a new Response with a different URL

Response data

- Response itself has the Body mix-in added to it, which permits access to the data of the Response
- Each of these methods returns a promise wrapped around the response data converted to the appropriate data type
  - **`Response.json()`**
  - **`Response.text()`**
  - **`Response.formData()`**
  - **`Response.arrayBuffer()`**
  - **`Response.blob()`**

Response properties

- Response properties give you read-only metadata about the Response
  - **`Response.status`**: status code (200, 404, etc.)
  - **`Response.statusText`**: status text (OK, Not found, etc.)
  - **`Response.ok`**: true if **`Response.status`** is between 200 and 299
  - **`Response.redirected`**: true if this response has been redirected

Response properties continued

- **`Response.type`**
  - **`basic`**: Normal, same-origin
  - **`cors`**: Normal, cross-origin
  - **`error`**: Network error, triggers the rejection handler of the promise
  - **`opaque`**: rejected cross-origin request
- **`Response.headers`**: The Headers object associated with the response
- **`Response.url`**: The URL of the response

Promises

- The **A** in Ajax stands for **asynchronous**, which means two things:
- First, after the request has been sent, JavaScript can continue to run code
  - JavaScript in the browser is mostly single-threaded, requests are handled by a separate browser thread
- Second, the code does not “know” when the request is going to come back
  - Responses are handled somewhat like events, in that they interrupt program flow when they come back
- Promises wrap around this uncertainty and provide an easy-to-use API to register code which will handle eventual responses

Chaining promises

- Promises are chainable
- Any call to **`then()`** returns a promise
- Let's call this a second-order promise
- The promise generated by the **`fetch`** promise is the first-order promise
- The second-order promise waits for the first-order promise to finish, and then runs
- The second-order promise is passed whatever the first-order promise returns
- And so on, and so on

More on promises

- Promises “wrap around” a data payload
- `p1.then( ( data ) => { ... } )`
- You cannot return data from outside the promise
- But you can return it to the next chained promise!
-
```
let p2 = p1.then( ( response ) => {
  return response.json()
} );
p2.then( ( jsonData ) => { ... } );
```

Promise tracks

- From a given promise:
  - throw error -> next promise is a failure / error / rejected
  - return rejected promise -> next promise is rejected
  - return nothing -> next promise is a success
  - return anything other than a rejected promise -> next promise is a success
  - no handler -> Falls through to the next promise
- Note that these hold for both rejection and resolved handlers
- Your error handlers should always do one of the following:
  - Re-throw the error
  - Throw a new error
  - Return a rejected promise
- Unless you intend to hide or suppress the error!

Handling rejection

- It is typical to append a catch-all error handler to the end of a promise chain
-
```
fetch( ... )
   .then( ... )
   .then( ... )
   .then( ... )
   .catch( rejectionHandler )
```
- This way, any errors from the original **`fetch()`** call or any of the subsequent resolved handlers can fall through to **`rejectionHandler()`**

Demo: Promises and fetch()

- File: **demos/src/fetch-promises.js**
- From the demos folder, run
  `npx webpack src/fetch-promises.js`
- Then navigate to **http://localhost:8000/demos** to see the results

Options

- **`method`**: The request method
- **`headers`**: Customized headers (as Headers)
- **`body`**: Any body that you want to add to the request
- **`mode`**: **`cors`**, **`no-cors`**, **`same-origin`**
- **`credentials`**: Request credentials: **`omit`**, **`same-origin`**, **`include`**

More options

- **`cache`**: Cache mode for the request
  - **`default`**, **`no-store`**, **`reload`**, **`no-cache`**, **`force-cache`**, **`only-if-cached`**
  - These map to the Cache-Control header
- **`redirect`**: Redirect mode to use
  - **`follow`**: follow automatically
  - **`manual`**: handle manually

Sending data

- For a GET request, append data directly to the URL
- For a POST, PUT, or PATCH, append data to the body of the request
- Use a JavaScript **`FormData`** object for form data
- Run a JavaScript object literal through **`JSON.stringify()`** to pass JSON

Exercise 8: Promises and Fetch

- File location: **exercises/ex-08**
- Start in **exercises/ex-08/ex-08-index.js**
- Follow the instructions there
- When you are ready to test your code:
  - Change directory to the **exercises** folder
  - enter
    ```
    npx webpack --watch ex-08/ex-08-index.js
    ```
  - Using a browser, navigate to **http://localhost:8000/exercises** to check the results
- If you run into trouble and cannot solve it yourself, ask your instructor for help

Conclusion

React and Redux

Credits and copyright

Introduction to React

Written by John Paxton

for Speeding Planet

http://speedingplanet.com

Flight check

- ➢ Can you see my screen?
- ➢ Can you read my code?
- ➢ Can you hear me?
- ➢ Any distractions?

Schedule

- Morning break
- Lunch break
- Afternoon break

Introduction to React

React Chapter 1

Introduction and Setup

Chapter preview

- Starting quickly
- Using create-react-app
- Creating a “Hello, world” component
- Testing our component

Starting quickly

- Our objective is to get into the guts of React quickly
- In the next chapter, we will talk about React itself
- For now, we want to get to writing code quickly!

Starting quickly, continued

- Do the following, if you have not already:
- Open a command prompt or terminal window
  - On Windows, change directory to somewhere near the root of the drive you want to use
- Use Git to clone the Classfiles repo:
  Ask your Instructor for the repo
- Change directory to the newly created **sp-react-class** directory
- Run `npm install`

Starting the server

- Windows: **`start npm start`**
- Macs (assuming bash shell): **`npm start`**
- The server will spin up, and your default browser should open soon at **http://localhost:3000** with a welcome page
- That's a React app!
- That was easy!
  - Although, to look at it, somewhat… underwhelming

Using create-react-app

- Facebook (React's creators) provides a utility for creating React applications from scratch
- Called, conveniently, **create-react-app**
- It takes care of all of the work of setting up a server, watches, and deployments for you
- This starter application is a slightly modified version of using create-react-app directly
- It frees us to focus on writing React code without worrying about setup issues

Side note: Creating applications

- You could use create-react-app to build your own application!
- Change directory one level up from **sp-react-class**
- Enter the following
  - **`npx create-react-app first-react --use-npm`**
- Change directory into **first-react**
- Run **`npm start`**
- The default application will pop up on your browser!

Building an application

- The “application” we currently have does not, actually, use much React
  - We should remedy that
- In the following exercise, we will add the basic files and code needed to create a “Hello, world” React app
- Usually, exercise directions will be in the files for the project
- But in this case, we should look at the details of a basic React application, so we will talk about the steps of the exercise here in the workbook

Overview

- At its most basic, React allows you to design custom HTML tags
  - This is a *gross oversimplification*, but will do for the first exercise
- We need to do two things:
- Write a custom tag
- Hook it up to the browser's Document Object Model

Exercise 1: Setup

- Ok, let's set up the application so that we can build it more or less from scratch
- First change directory back into sp-react-class if you have not done so beforehand
- Run this command:

```
npx gulp start-exercise --src ex-01
```

- Enter it exactly as above
- This will clear out the src/ directory and prepare it for the first exercise

Exercise 1: Hello world and more

- Let's start by building a simple custom tag, called a component
- In the **src** folder in **sp-react-class**, create a file called **App.js**
  - Component names are capitalized
  - Files should contain one and only one component
  - The filename should be the same as the name of the component
- This will contain our new component **`<App/>`**

Exercise 1: Defining a component

- The component can be written as a function
  - Later on, we will write components as ES2015 classes
  - You can mix and match, or define rules for when to use function-based components vs class-based components
  - Or you can just use one or the other
- Right now, the function takes no arguments
- Instead, we will have it return the HTML content of the component

Exercise 1: Returning HTML

- Our React components will return HTML in the form of **JSX**
  - It actually does not stand for anything!
  - It is a mix of JavaScript, XML, and HTML
  - We will look at the syntax in more detail later
- Two important points now:
  - Use **`return ( … )`** to allow for statements which span lines
  - You must return **one root element**
  - Your root element can have as many sub-elements as you want
  - But there must be one root element

Exercise 1: The React module

- Our component is a function which is part of an ES2015 module
- The module must import the React module to work
- Specifically, it must import **`React`** from **`react`**

Exercise 1: Our new component

```
import React from 'react';

export default function App() {
  return(
    <h1>Hello, world!</h1>
  )
}
```

Exercise 1: Tying the component to the DOM

- Next we need to tie the component to the DOM
- For the rest of the course, we will usually create and use components, not worrying about this step
- This is, then, a one-time step to connect the React application to the Document Object model of the browser
- The file that does this is usually called **index.js**
  - This is a convention, not a requirement

Exercise 1: Creating index.js

- index.js is an ES2015 module
- index.js needs to import three modules:
  - **`React`** from **`react`**
  - **`ReactDOM`** from **`react-dom`**
  - **`App`** from **`App`**
- Importing **`React`** should be self-explanatory
- Importing **`ReactDOM`** provides the function that will add custom components to the DOM
- **`App`** is the custom component in question

Exercise 1: index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

Exercise 1: Connecting to the DOM

- **ReactDOM** has the `render` method which takes two arguments:
- The HTML or JSX to render (almost always JSX)
- A reference to an existing DOM node to render to
- If you look in **public/index.html**, you will find the following node:

```
<div id="root"></div>
```

- This is the target node for `ReactDOM.render`

Exercise 1: Summary

- Create a component
- Create **index.js**
- Add a target element to **index.html**
  - This was already done for us
- Tie the component to the target in index**.js**
- Examine code in a browser!

React and TDD

- Throughout this course, we will place a special emphasis on testing
- We will not be using strict TDD
  - No test first, then right code
  - That is good for code, but not great for learning concepts!
- But we will write tests (failing and successful!) for the concepts we learn

Testing React

- The Facebook team created the Jest framework to test React
- Jest incorporates a variety of testing features from various other frameworks
- And Jest is intended to be the official test framework for React
- We will use Jest to test all our React code
- Jest is included as part of any application created with create-react-app

Exercise 2: Testing “Hello, world”

- Let's start with a basic test
- There are two parts of a Jest test:
- The test itself, created via the **`test`** method
- Expectations within the test, a combination of the **`expect`** method and a **matcher**
    - You can have multiple expectations in one test
    - If you have more than 3-4 expectations, you should probably check to see if you can break this one big test into several smaller ones

Exercise 2: Creating a test

- Create a file in the **src** folder: **App.spec.js**
  - Jest automatically picks up files that contain the string **`.test`** or **`.spec`** in their names
  - Or any tests in the __**`tests`**__ directory
  - This is configurable, though not in create-react-apps
- Add the code on the next slide to **App.spec.js**

Exercise 2: A basic test

```
test( 'Adds 2 and 2 to equal 4', () => {
  expect( 2 + 2 ).toBe( 4 );
} );
```

Exercise 2: Running the test

- Open a new command prompt or terminal window
- Change directory to the **sp-react-class** directory
- Run **`npm test`**
- You should see results indicating that the test was a success!
- You can leave this window open, it will continue to watch your test(s) for changes, and re-run those tests when they change

Exercise 2: Testing a component

- That was a nice test but not, you know, realistic
- We would like to test our actual component
- We will write two more tests
- One to see if the component loaded correctly
- Another to see if the component has the right content

Exercise 2: Does it load?

- How can we test a component to see if it loads?
- When we work with a component, we tie it to the DOM and then render it
- So we need a Document Object Model to work with, and a way to render a component to it
- Oh, and a component, too, but we have that in App

Exercise 2: Rendering to the DOM

- Creating a DOM is actually simple: just use the DOM interface
- **`document.createElement`** will return a reference to a DOM element
- How did we render to the “real” DOM in our application?
- We used **`ReactDOM.render`**
- Which we can use again here in our test
- Finally, we will not have an expectation
- The fact that the component renders to the DOM without error **is** the test

Exercise 2: Adding a test

- You can add the code from the next slide to **App.spec.js**
- Leave the test that we already created in the file
- Add the imports at the top
- And the new test either before or after the first test
  - Note that the new test uses the **`it`** method instead of the **`test`** method
  - They are interchangable (**`it`** is an alias to **`test`**)
- The **`npm test`** window should pick up the changes and re-run the test for you automatically

Exercise 2: Test rendering

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

// Leave our first test here

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

Exercise 2: Test content

- Great, we have a basic test, and we can test that the App component actually renders to a DOM element
- But do we know if it renders useful information?
  - Given, our App component is hard-coded at the moment
- How can we test for the content of the element?
- We could use the DOM to retrieve the content of the created <div>
  - But this could get tricky, depending on rendering time, methods used to access the text of the DOM element, etc

Exercise 2: Using Enzyme

- Instead, we will add a test helping utility called Enzyme
- Enzyme, created and maintained by AirBnB, makes it easier to work with React components
- We will use Enzyme to shallowly render the App component
- And then test its content

Exercise 2: Shallow rendering

- Enzyme exports a method, **`shallow`**, which takes a component object as an argument
- It returns a **wrapper** around the component that lets us test it as if it were rendered
- In particular, we are interested in the **`wrapper.text`** method, which pulls out a String representation of all of the text nodes in the rendered component

Enzyme adapter

- As of Enzyme 3.0, it requires an adapter to work with a particular version of React
- The adapter can be loaded like so

-
  ```
  import Enzyme, { shallow } from 'enzyme';
  import Adapter from 'enzyme-adapter-react-16';

  Enzyme.configure({ adapter: new Adapter() });
  ```

Exercise 2: Using Enzyme

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import Enzyme, { shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
Enzyme.configure({ adapter: new Adapter() });

// Tests 1 and 2 here

test('Contains "Hello"', () => {
  const wrapper = shallow(<App/>);
  expect(wrapper.text()).toMatch(/Hello/);
});
```

Exercise 2: Analysis

- We added an **`import`** line to bring the **`shallow`** method in from enzyme
- We also added a test which created a wrapper around a shallowly rendered component
- The test looked at the text of the component to see if it included what we expected

Exercise 2: Summary

- Testing React is very similar to writing React
- Either use React directly, or add in helper frameworks like Enzyme to test components
- Render the component you want to test, and then check its properties and state to see if it is acting the way you expect it to
- We will expand our testing capabilities as the course continues

Conclusion

React Chapter 2

React Architecture

Chapter preview

- About React
- Why React?
- Passing data
- Our environment
  - Server
  - Transpiler
  - Watching files
- Where do we go from here?
- The demos site

About React

- React was created by Facebook and carries the open source BSD license
  - It is used extensively by Facebook, Instagram, Netflix and others
- React is “[a] declarative, efficient, and flexible JavaScript library for building user interfaces.”
  - According to the GitHub repository for React
- React is a View library, it does not provide nor is it opinionated about models or controllers
- It is wholly and solely concerned with rendering the UI

About React, continued

- Again, from the GitHub repo, React is:
  - **Declarative**: React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. Declarative views make your code more predictable, simpler to understand, and easier to debug.
  - **Component-Based**: Build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.
  - **Learn Once, Write Anywhere**: We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React info sheet

- Home page: https://reactjs.org/
- Docs: https://reactjs.org/docs
- Github repo: https://github.com/facebook/react
- Version: 16.x (16.6.1 as of November of 2018)
- React devtools: https://github.com/facebook/react-devtools
- React tool suggestions: https://reactjs.org/community/debugging-tools.html

Why React

- What are the reasons for using React? Put another way, what does React do best?
- React is fast: perhaps one of the fastest UI renderers extant
- React is simple: React itself is not very complicated and does not aspire to be anything more than a view library
- React is extensible: Nonetheless, it is easy to extend React with other tools (like Redux) or to use it in a variety of different circumstances, situations, or environments
- React is component-based: React is aligned with the future of web application architecture in general (component-based vs MVC-style)

Component-based applications

- React promotes development of small, extensible, loosely coupled components as the building blocks of applications
- You do not truly build an "application" per se in React
  - You build the presentation of the application
- Use and re-use components to render various aspects of your application
- Do not worry about rendering or updating, as you can let React manage that part

Components vs MVC

- There is not a “right” way to do client-side applications
- The first generation of JavaScript application frameworks (think AngularJS and Backbone) leaned heavily on the MVC pattern for inspiration
- In the context of the web, this has some shortcomings
  - Web application need to update their UI often
  - UI updates are costly and should be optimized
  - Clients have no persistent data connection
  - Clients must maintain their own state, but also reconcile that state with server state

Components vs MVC, continued

- The second generation of client-side applications (React, Angular, Vue, others) relies on a component-based approach
- Rather than strict definitions of Model, View, and Controller, components represent parts of the View
- Components do not care about where they get the Model they display
- Components do not know about the concept of a Controller, either
- As application designers, this gives us flexibility in figuring out the M and C roles, while leaving React to manage the View

Functional-style components

- In the first chapter, we created a component from a function
- Components can be created from functions
- They are limited in their capabilities by the limitations of a function
  - No methods
  - No constructor
  - Limited arguments
  - No lifecycle overrides
- But they are lightweight, easy to create, and easy to manage
- In the future, React *may* optimize functional components

Class style components

- Components can be created from ES2015 classes via inheritance
- Import **`Component`** from **`react`**
- Have the class extend **`Component`**
- Provide a **`render`** method
  - In functional components, the function itself **is** the render method
- Class style components have all of the benefits of classes
  - Extensible
  - Full lifecycle of events
  - Break out functionality into methods
  - Constructor
  - And more

Class vs functional components

- So which should we use?
- The answer is, what does your component need?
- Some argue for using functional components when the component is stateless
  - And class-based components when the component is stateful
  - We will see a variant of this argument later in the course
- If you need features only accessible in class components, use those
  - Otherwise, prefer functional components
- In this course, we will use functional components unless we need the features of a class-based component

Demo: Class-based components

- Here is the component from Exercise 1, re-rendered as a class-based component
  - This has no implications for testing, so it should continue to run fine
  - You can find this in **exercises/ex-02/solution/src/AppClass.js**

```
import React, {Component} from 'react';
export default class AppClass extends Component {
  render() {
    return (
      <h1>Hello, world!</h1>
    )
  }
}
```

Passing data

- So far, we have built and tested a pretty basic component
- Let's expand the capabilities of the component by passing data to it
- Passing data to a component in React is surprisingly easy
- Inbound data is sent via attributes
  - **`<MyComponent someInput="someValue"/>`**
- In the component, you can access the data as a field called props
- Most components are written as **`(props) => { … }`**

Using data

- You can use passed data in your JSX as **`{props.propertyName}`**
- The props collection contains any and all information passed into this component as an attribute
- The props collection is **immutable**
  - Primitives cannot be changed at all
  - Object references can have their state changed, but not the reference itself
- Think of a component as a function: props are the input, HTML is the output

Component data

- What about component data?
  - Data that belongs to the component itself, but is not passed in
- Any data in the component can be accessed by its variable name
  - Declare a variable
  - **`let x = 10;`**
  - Use it in the JSX of the component
  - **`<span>x is equal to {x}</span>`**
- Later, we will use a special variable for this component data

Class-based data

- Accessing data in a class-based component is done differently
- If the class does not have a constructor, properties are available as **`this.props`** throughout the class
- If the class does have a constructor, it should have one argument, **`props`**, which is also passed to the **`super`** constructor
- Since JavaScript classes do not have fields, there is no equivalent to the function variables we saw with functional components
  - But do not worry, we will not, use the functional variables all that often

Demo: Properties

- Look in **sp-react-demos/src/demos** at the following
  - **`ClassProps`**
  - **`ClassPropsContainer`**
  - **`FunctionalProps`**
  - **`FunctionalPropsContainer`**
- Navigate to **http://localhost:3001/demos** and try the demos
  - Functional Component with props
  - Class Component with props

Demo, explained

- Start with **`FunctionalPropsContainer`**
- It is a very simple wrapper around **`FunctionalProps`**
  - This is a pattern we will see many times
- **`FunctionalPropsContainer`** creates an instance of **`FunctionalProps`** and passes it a name property
- In **`FunctionalProps`**, the name property is accessed as props.name
- Things are relatively similar for **`ClassPropsContainer`** and **`ClassProps`**

Exercise 3: Passing data

- We will update our application with custom headers and footers
- The **`CustomHeader`** and **`CustomFooter`** components are mostly concerned with CSS styling
- Though they take arguments to determine their content
- Your instructor will walk you through using **gulp** to set up for this exercise
  - Gulp is a task runner which runs on Node JS
- Run **`npx gulp start-exercise --src ex-03`**

Our environment

- We are using an application created by **create-react-app**
- create-react-app makes it easy to prototype a React application without having to worry about many decisions related to setting up the environment
- It provides a web server, file watchers, ES2015 implementation, test framework, and code linting
- This application has had **`npm eject`** run on it to create a standalone application
  - We cannot reverse this process, though
  - An ejected application is no longer a create-react-app application

Important parts of the application

- In this class, we do not worry to much about the application environment
- But in the world outside of class, you should think about these parts of your environment:
- Web server (Node JS + Express? Java + Tomcat? Windows + IIS?)
- Test framework (probably Jest, possibly with helper frameworks)
- ES2015 implementation (Most likely Babel)
- Task runner (Gulp, Grunt, or Maven, or Gradle, etc)
- Code packaging (Webpack, Browserify, possibly others)

Where do we go from here?

- What is the plan?
- The concept for the class is that we are building a consumer banking application
  - Similar to what you would see when you log in to your bank to look at your checking account
- Some of the application is complete and is hosted on a demo site
- Other parts of the application will be our responsibility
  - Mostly having to do with the payees section

The demo site

- If you have not already, clone the repo at **https://github.com/speedingplanet/sp-react-demos**
- Change directory into the sp-react-demos folder
- Run **`npm install`**
- Run **`npm start`**
- The demo site should come up, probably at **http://localhost:3001** or **http://localhost:3002**
  - This may depend on what else you have running

Transactions as demos

- The demos web site covers the Transactions part of the application
- It includes searching, list, and detail views, as well as functionality for adding and editing transactions
- It can be configured to use data locally, (i.e., with no external server), via Ajax, or via Redux
  - We will see the Redux-ified version later in the course

REST server

- We also have a RESTful server available to us
- You can start it from **sp-react-class** by executing **`npm run rest`**
- The server has several RESTful endpoints
  - tx (transactions)
  - payees
  - accounts
  - people
  - categories
  - staticData
- It is implemented by **json-server**, an npm package

Summary and conclusion

- React was created by and is maintained by Facebook
  - With many outside contributions
- React is a client-side view library
- React implements component-based architecture, rather than strict MVC
- React itself is agnostic about other tools in the web development environment
  - Possibly excepting Jest as the test framework
- As designers, we should be aware of the choices of other tools we will need in the environment, external to React

React Chapter 3

Basic React Components

Chapter preview

- ➢ Introduction to JSX
- ➢ Classes and style
- ➢ Snapshot testing
- ➢ Using child components and content
- ➢ Conditionally displaying data

Introduction to JSX

- Whether using functions or classes to create components, we use JSX to generate HTML
- JSX is a mix of JavaScript, HTML, and XML
- It follows XML's rules about being well-formed
  - One, single root element
  - Tags must be **`<balanced></balanced>`** or **`<standalone />`**
  - Tags must match their nesting **`<i><b></b></i>`** not **`<i><b></i></b>`**
  - Tags are case-sensitive

JSX and React

- A few more rules for working with React and JSX
- The first letter of the name of the component is capitalized
  - Especially since it is often a class as well
- Attributes should be bound with quotation marks
  - Single or double does not matter, but be consistent
- But attributes and other information can be substituted with { }
  - **`<MyComponent someAttr={someValue} />`**
- Whenever you use JSX, React **must** be in scope (imported into the current ES2015 module)

JSX and JavaScript

- We can add arbitrary JavaScript to our JSX by enclosing it in curly braces
- This is true for attribute values, code between elements, anywhere in our JSX we like
  - But we still have to follow the XML rules and keep our code well-formed
  - In practice, this means that our JavaScript is somewhere under the root element of our JSX
- The JavaScript of JSX is not a sub-set or a DSL, it is real JavaScript
- Write to whatever version of ECMAScript your deployment will support

JSX and context

- In JavaScript, context is very important for resolving variables
- JSX is no different
- The context for any variables in your JavaScript expressions in JSX is the context of the method (usually render) creating the JSX
- Variables do not cross scopes in JSX
- If your component is the child of another component, your JSX does not have access to the parent component
  - Or any siblings, for that matter
  - We will see soon how to pass information among components

JSX syntax

- A few things are different in JSX syntax
- Comments out a block with **`{/* <Block /> */}`**
- Use a string literal in quotes **`{ 'string literal' }`**
- Values like **`false`**, **`undefined`**, **`null`**, and **`true`** are valid, but do not render out anything in the page
- Do not forget to return a single root element
- Do not forget that the **`return`** statement can wrap lines by using parentheses

JSX and child components

- JSX also allows you to render child components
- Given this structure

```
<Parent>
  <Child>
    Some text
  </Child>
</Parent>
```

- The component **`<Child>`** will be in the **`props.children`** property of **`<Parent/>`**
  - Likewise, the text **`Some text`** will be the props.children of **`<Child/>`**
  - In a class-based component, refer to **`this.props.children`**

Classes and style

- CSS and React have some particular rules about interactions
- First, when specifying a CSS class or classes in JSX, use the **`className`** attribute instead of **`class`**
  - **`class`** is a reserved word in JavaScript, preventing its use in JSX
- Otherwise a **`className`** is a property like any other, set it in JavaScript and assign it a literal or a variable
  - **`className=“foo bar baz”`**
  - **`className={myClassList}`**

Style attributes

- Style attributes must be JSX assignments
- **`style=“color:blue”`** is not valid in an element
- Use a JavaScript object literal to specify an in-line style

-
  ```
  let myStyle = { color: 'blue' }
  …
  <li style={myStyle}>…</li>
  ```
- Or specify. an object literal in-line
- **`<li style={ { color: 'blue' } }>…</li>`**

Demo: JSX examples

- Look at **JSXExamples.js** in the demos folder in the **sp-react-demos** project
- Pay attention to the various examples of JSX usage
- In particular, note the use of **`className`** and style objects with respect to CSS

Exercise 4: Using JSX

- In this exercise, we will conditionally display data, as well as use **`props.children`** to render content
- Objectives:
  - Use inline JavaScript in our JSX to conditionally display information
  - Use the **`props.children`** feature to render out child elements of the parent component without knowing them in advance
  - Style elements according to information in **`props`**
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-04`**
- If you have problems with the exercise, ask your instructor for help

Results testing

- Our last two exercises have focused on rendering out content to the DOM
- We would like to write some tests which can focus on the results of rendering a component
- Rather than test a small part of the component (which is still a useful test, of course), we would like to look at the component rendering tree as a whole
- Given a component, particularly one with child components, can we ensure that it is still rendering the way we expect after a change?

Snapshot testing

- Testing the entire results of a rendered component is an ideal use case for snapshot testing
- The name comes from the practice of taking a snapshot of rendered content, and then making changes
- After the changes, a second snapshot is taken and then compared to the original snapshot
- The test passes if the two are the same
- Snapshot testing ensures that, despite changes, updates have not broken your component's UI functionality

Creating snapshots with Jest

- Snapshots are easy to create
- Import **renderer** from **react-test-renderer**
- Use renderer to create a snapshot and serialize it to JSON

  -
    ```
    const snap = renderer.create(
                   <MyComponent/> ).toJSON()
    ```
- Use the matcher **`toMatchSnapshot`** in your expectation
  - **`expect(snap).toMatchSnapshot()`**
- Snapshots are put into a folder called **`__snapshots__`** in the same folder as the test

Snapshots

- When you first run a test which uses renderer, it creates a snapshot file
  - It is in human-readable JSON
- Subsequent test runs are compared to this file
  - If the snapshots match, the test is successful
- If you need to regenerate snapshots, invoke jest like so
  - **`jest --updateSnapshot`**
- Commit your snapshots to your VCS along with other test code

Demo: Snapshot testing

- In **sp-react-demos**, there is a demo called **`TreeComponent`**
- It is a simple component which renders out some children and then exits
- In the **tests** folder, you will find a test for **`TreeComponent`**, as well as the **`__snapshots__`** folder
- Look at the file with your instructor, following the instructions therein
- You will see how to run a successful test, as well as how to break the test

Exercise 5: Snapshot testing

- We will use snapshots to test the code from our last exercise
- Objectives:
  - Generate snapshots in our tests
  - Ensure that our tests pass when making minor updates to the code
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-05`**
- If you have problems with the exercise, ask your instructor for help

Conditionally displaying elements

- JSX and JavaScript can be used to conditionally display entire elements
  - Rather than just changing the content of elements
- Remember that elements are values in JSX, just like numbers, strings and so on
- Assign an element to a variable conditionally
- Then display the variable in the **`return`** statement of your component's rendering function
- Or, use a logical operator like **`&&`** or **`?:`** inline

Demo: Conditionally displaying data

- Check out **ConditionalDemo.js** in your **demos** project
- **ConditionalDemo.js** actually defines several components in one file
  - We would not do this in the real world, of course, but it is convenient for a demonstration like this
- Note the various ways that **ConditionalDemo** uses to print a component depending on a condition

Exercise 6: Conditionally displaying data

- This is the last of our exercises to experiment with our "Hello, world" home page
- In this exercise, we add conditional component display to our bag of tricks
- Objectives:
  - Depending on whether the user is logged in, display a ”Log In” button or a “Logged in as …” banner
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-06`**
- If you have problems with the exercise, ask your instructor for help

Conclusion

- So far, we have been working with simple React components, trying things out and experimenting
- We have used JSX to render content
- Sometimes conditionally!
- And also taken snapshots of that content for testing purposes
- In the next chapter, we will move over to a set of “real world” exercises

React Chapter 4

Event handling and state

Chapter preview

- Our real-world example: Payees
- Basic event handling
- Props and state
- Extracting components
- Inter-component communication
- Testing component state
- Using spies
- PropTypes

Our real-world example

- In the last chapter, we focused on a simple Hello, world example
- Now we are going to move to working with a more realistic topic: a consumer banking site
- Some of the site has been implemented over at the demo project
- We are in charge of implementing the Payees portion of the site
- Payees are associated with Transactions (tx) and Categories

A typical Payee object

```
{
  "id"         : 24,
  "payeeName"  : "Rodriguez Outfitting",
  "categoryId": 101,
  "address"    : "587 Ipobak Terrace",
  "city"       : "Alexandria",
  "state"      : "VA",
  "zip"        : "16097",
  "image"      : "/images/nature/6.jpg",
  "motto"      : "Operative maximized matrices",
  "active"     : true
}
```

Notes about Payees

- Payees are associated with categories through a category **id**
  - Categories have a **categoryName** and a **categoryType**
- Payees do not know about their Transactions
- But Transactions know about Payees
- Payees may have an image, but there is, at the moment, no image folder where that image is held

Exercise 7: Rendering a Payee

- Our first job is to create a **`PayeeDetail`** component
- This component will render the relevant portions of a Payee
- We will be using Bootstrap, a popular CSS library, to render the Payee as a panel
  - Bootstrap's docs can be found at **getbootstrap.com**
  - Specifically, we will use a Bootstrap Panel to render the Payee
  - http://getbootstrap.com/components/#panels

Exercise 7: Getting data for the Payee

- Where can we get data from?
- Later on in the course, we will use the Fetch API to retrieve data from a remote REST server
- For now, we will include the data in our project directly
- Our **`PayeeDetail`** component can import **data/class-data.js**
- The class data module exports an object, **`payeesDAO`**, which provides access to Payees
- Ask for a Payee by **id** with **`payeesDAO.get(id)`**

Exercise 7: Rendering a Payee

- Now that we know how to access data, here are your tasks:
  - Build a **`PayeeDetail`** component
  - Get a Payee (payee #23 for instance)
  - Pass it into **`PayeeDetail`**
  - Render out its content
- Use gulp to load the class files for the exercise
  - **`gulp start-exercise --src ex-07`**
- If you have problems with the exercise, ask your instructor for help

What's next?

- The next feature on our to-do list for Payees is to allow users to page through a set of Payees
- What will we need to implement a simple paging toolbar?
- Next and previous buttons
- A set of data
- A way for the buttons to ask for the next or previous button in the set
- Let's dive in to React's version of event handling

Event handling and state

- So far our components have been static
- Charged with simply displaying the information passed to them
- In the real world, our components will need to react to user input, and manage information according to said input
- Reacting to user input is the province of event handling
- Managing that changing data comes under the heading of managing state
- We will look at event handling first

Event handling

- To implement event handling with React, add code in two places
- First, in the component, add an **event handling function**
  - Class-based and functional components handle this differently
  - Look at the following slides for details
- Tie an event to the handler in the JSX
- Given an event handler, **`clickHandler`**, defined in your component code
- Tie it to the component by adding the following attribute **`onClick={clickHandler}`**
- Note that to bind an event handler, we use **`{ }`** not quotation marks

Functional component event handling

- In functional components, define the event handler as a sub-function of the component function
- In the JSX for the event handler, bind the function directly
- **`<button onClick={clickHandler} />`**
- It is somewhat of a convention that the event handler for a DOM event **`foo`** is **`fooHandler`**

Class-based event handling

- Class-based event handling is somewhat more complex
- First, the handling function should be a class-level method (i.e., a sibling of constructor)
- Beyond that, there are complications to how to bind the event to the handler
  - ”Complications” from one perspective, “flexibility” from other perspectives
- See the next slide for details

Binding event handlers

- The typical form to bind a class-based event handler is
  `<button onClick={this.handleClick} />`
- Unfortunately, the **`this`** in **`this.handleClick`** will not be bound correctly without some assistance
- In the **`constructor`**, add this line:
  `this.handleClick = this.handleClick.bind( this )`
  - Yes, that looks weird
  - This rewrites the **`handleClick`** method of the component to always be bound to the current instance
  - Yes, that should happen automatically (maybe in a later version of React)
- This is the best, broad-use-case, most-efficient way to bind handlers

Other ways to bind event handlers

- **`<button onClick={ (event) => { … } } />`**
  - No need to bind this handler in the constructor
  - But it creates a new handler each time the component is rendered
  - This may cause extra re-rendering
  - Never use this in a loop, for instance
- **`handleClick = () => { … }`**
  - Use this at the class method level
  - Arrow functions do not re-bind the `this` keyword

Demo: Event handlers

- Look at **EventHandling.js** in the demos folder of the **sp-react-demos** project
- The code therein will walk you through several different event handling scenarios and styles

Exercise 8: Event handling

- The next three exercises have a common goal: Build a **Payee browser** where users can move through a set of Payees one at a time by clicking on **Next** and **Previous** buttons
- First, we will set up the buttons and attach event handlers
- Later on we will hook these up to data in the component
- Finally, we will use inter-component communication to choose which Payee to display

Exercise 8: Event handling

- Objectives:
  - Add two buttons to the **`PayeeDetail`** component, **Next** and **Previous**
  - Attach event handlers to the respective buttons
  - For now, when the event handler triggers, you can simply log the event to the console
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-08`**
- If you have problems with the exercise, ask your instructor for help

Props and pure components

- The **`props`** collection contains all the data passed to this component
- It is immutable, and you should not attempt to change it
- Components which use only **`props`** are called **pure components**
  - And, since they usually use the functional style, they are often known as pure functional components
- Pure components (like pure functions)
  - Do not modify their input
  - Have no side effects
  - Given the same input, always return the same output

Why are pure components useful?

- Why do we care about pure components?
- Pure components are easier to test
  - Standard inputs, no mocking
  - No side effects to worry about
- Pure components are often easier to reason about
  - The inputs are known
  - The expected output is known
  - The work of the component is tightly focused
- Pure components may be optimized by JavaScript and React
  - Depending on the JS engine and future versions of React

Component-level state

- But what if my component needs to manage internal information?
  - Which is commonly called **state**
- React has a different data collection for malleable information within the component: **`state`**
- The **`state`** variable is not available to functional components, only to class-based components
  - In the controller as **`this.state`**

Working with state

- The state variable should be initialized in your component's constructor
  - **`this.state = { … }`**
- Initialize state as an object literal
  - It can have nested objects as needed
- Access state as **`this.state.variableName`**
- Anywhere you modify state, use the **`this.setState`** function
- Pass it the modified state as an object literal
  - **`this.setState( {name: 'John'} )`**
- Do not modify state with direct assignments!

State and setState

- Modifying state with **`setState`** tells React that it may need to update this component
- **`setState`**'s job is to determine whether there was a change which affects the DOM
- We will go into the lifecycle details later, but the short version is that **`setState`** may trigger a re-run of the **`render`** method on the class
- Because **`setState`** may trigger DOM updates, it may act asynchronously, and it may choose to batch updates, if they are coming rapidly
- Never modify state without using **`setState`**!

Modifying state

- Always use **`this.setState`** to modify state
- Do not assign to **`this.state`** directly
- DO NOT DO THIS:
  - **`this.state.name = 'John'`**
- Modifying **`this.state`** directly prevents React from testing the data to see if it needs to update the DOM
- Your changes will not be reflected in the UI
- Use **`this.setState`** to change state!

Demo: Using state

- We will look at a component which uses **`this.state`** and **`this.setState`** to manage its state
- Under **sp-react-demos**, find **src/demos/UsingState.js**
- Your instructor will walk you through the details of this demonstration

State vs props

- So what should be **state**, and what should be **props**?
- Guidelines for state:
  - Belongs to this component
  - Original, cannot be calculated from other available values (other state, props, etc.)
  - Changes over time
- Props can become state, it's unusual but not rare
- Think of doing so as making a local copy of data to modify

Exercise 9: Working with state

- Part 2 of our Payee pager exercises
- Objectives:
  - Load a list of Payees
  - Use **`state`** to track the currently displayed Payee
  - Use event handlers to browse through the set of Payees
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-09`**
- If you have problems with the exercise, ask your instructor for help

Multiple components

- So far, we have kept our application to two components: **`App`** and **`PayeeDetail`**
- But our **`App`** may eventually have other topics
  - Transactions, Categories, etc
- And we will definitely have other components under Payees
- React is about building reusable, loosely coupled components
- So we should refactor our application to take that into account

Presentational and container components

- Adapted from https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
- When working with components, it is useful to divide them into two types: purely presentational, and containers
- Presentational components do not usually include data access or business logic
  - Their job is to display the content handed to them
  - Usually also pure functional components
- Container components may include some presentational aspects
  - But their job is more likely to manage child components, be they presentational or containers themselves

Extracting components

- So, we are going to split out some responsibilities
- We will create a **`Payees`** component, which will be the container component for the Payees portion of our application
- The **`Payees`** component will contain the **`PayeeDetail`** component
  - The **`PayeeDetail`** component which is a presentational component
- **`Payees`** will be responsible for holding on to the array of Payee objects

Parent to child communication

- We have seen values passed into elements before
  - **`<HelloWorld name={firstName}/>`**
- We can do the same with our **`Payees`** to **`PayeeDetail`** communication
  - A parent component can easily pass information to a child component
  - In this case, **`PayeeDetail`** will be passed a selected Payee
  - If we want **`PayeeDetail`** to re-render if and when the selected Payee changes, what should we do?
  - Have the selected Payee be part of **`Payees`**' state!
  - When we change the selected Payee in **`Payees`** via **`setState`**, we will trigger a re-render in **`PayeeDetail`**

Child to parent communication?

- What if a child component wants to send a message to a parent component?
- Think of an edit form, purely presentational, which wants to inform a container parent about changes to its content
- For our case, think of the **`Next`** and **`Previous`** buttons
  - They do not belong to a **`PayeeDetail`** component
  - They do not belong to the **`Payees`** component
  - We will probably create a component for them
  - But how can we customize the behavior of the buttons?
  - How can we make our **`BrowserButtons`** component flexible?

Demo: Custom events

- In **sp-react-demos**, under the **demos** folder, look at **CustomEvents.js**
- Note that there is a basic counter variable and a set of buttons
  - How familiar!
- When the buttons are clicked, they emit custom events, **`onIncrement`** and **`onDecrement`**
- The parent **`Counter`** component passes in an event handler for the custom events

Inter-component communication

- The answer is custom events
- Create a custom event on a child component
- When a parent component uses a child component, the parent can pass in an event handler for the custom event
- When the child fires the custom event, the parent's event handler fires
  - Often, the child fires the custom event in response to a DOM event
  - Click on a button in the child, fire a custom event in the **`onClick`** handler
- The child component is sending a message to the parent anytime a custom event occurs

Exercise 10: Extracting components

- Objectives:
  - Extract some code into the **`PayeesContainer`**, **`PayeeDetail`**, and **`BrowserButtons`** components
  - Add custom events to **`BrowserButtons`**
  - Refactor code to handle **`BrowserButtons`**' custom events
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-10`**
- If you have problems with the exercise, ask your instructor for help

Testing event handling

- We have gone through several exercises to build our small-scale Payee browser
- And we do not have any testing to go with it!
- We should remedy this
- We could use basic testing and snapshot testing for some of our testing needs
- We will need to simulate user behavior, specifically clicking on a button, to have thorough testing

Testing events

- The Enzyme test library allows you to simulate events on a component
- After wrapping a component, use the simulate method, passing in the name the the event to fire and any other relevant arguments
- **`wrapped.simulate('click')`**
- Enzyme actually finds the corresponding property (**`onClick`** in this case) and fires it
- This is not a proper DOM event and does not propagate
  - Which is why it is called “simulate”

Checking state and props

- Enzyme-wrapped components have access to their respective state and props properties
- **`wrapper.state().key`**
  - Also **`wrapper.state(key)`**
- **`wrapper.props()`**
  - Returns a collection of the current **`props`** for this component
- **`wrapper.prop(key)`**
- Use these to test that the component is in an expected state after simulating an event

Shallow and Full rendering

- When we first talked about enzyme, we used it to render a component shallowly
  - At the time, we did not have parent-child component relationships, so shallow rendering made sense
- If we are to test our nested elements, we may have to fully render them with the mount method
- Enzyme makes **`mount()`** available to fully render an element to the Document Object model

Full rendering requirements

- Fully rendering a component requires an actual working DOM
- You could run your tests in a browser, or
- You could use a simulated DOM like **jsdom**
- "jsdom is a pure-JavaScript implementation of many web standards, notably the WHATWG DOM and HTML Standards, for use with Node.js. In general, the goal of the project is to emulate enough of a subset of a web browser to be useful for testing and scraping real-world web applications."
  - From the jsdom GitHub page

Demo: Simulating events

- In **sp-react-demos**, under the **demos** folder, look at **CustomEvents.js** as well as **tests/TestingEvents.spec.js**
- The component in **CustomEvents.js** has some button-based interactions
- In **TestingEvents.spec.js**, we simulate these events and then check the state and props of the relevant components to ensure they have the correct values

Spying on changes

- Firing an event may have concrete changes
- But it may also have subtle changes
- We want to be able to see what code ran when an event is fired
- The **Sinon** library enables spying on methods
- We can use Sinon to spy on an event handler to ensure that it was called
  - Rather than looking for the effects of an event handler having been called
- http://sinonjs.org

Working with Sinon

- Sinon allows you to create spy methods
  - **`let callback = sinon.spy()`**
- Pass a spy method into an event handler
- Simulate an event
- Check to see if the spy method has been called
- **`expect(callback.called).toBeTruthy()`**
- **`expect(callback.callCount).toBeGreaterThan(0)`**

Spying on existing code

- Sinon can wrap existing functions
- The functions will … function normally
- But Sinon can track how many calls there have been to the handler, with what arguments, and so on
- **`let spy = sinon.spy(clickHandler)`**
- **`let spy = sinon.spy(objOrWrapper, 'method')`**
- If Sinon is wrapping a **`mount`**ed component function (rather than a shallowly rendered one), call **`wrapper.update()`** to activate the spy
  - If you do not call **`wrapper.update()`** the spy is never invoked properly

Demo: Testing with spies

- In **sp-react-demos**, under the **demos** folder, look at **tests/TestingSpies.spec.js**
- The code in **TestingSpies.spec.js** is still testing **TestingEvents.js**
- But instead of testing state and props, we are using Sinon to spy on function calls

Exercise 11: Testing components

- Objectives
  - Test our **`PayeesComponent`**, **`PayeeDetail`**, and **`BrowserButtons`** components
  - Use the **`state()`** and **`props()`** methods to test that the components have the proper state and props
  - Use spies to ensure that event handling functions were called correctly
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-11`**
- If you have problems with the exercise, ask your instructor for help

Conclusion

React Chapter 5

Type checking and lists

Chapter preview

- Type checking
- Using **`PropTypes`**
- Lists of data
- Sorting a list

Type checking

- JavaScript is neither strictly nor strongly typed
  - There are numerous advantages and disadvantages to this approach
- Under some circumstances, you may want to have a certain amount of type checking available
  - Development mode
  - Working with components
  - When you need to know the various arguments required for a component
- Facebook defines a lightweight type checker for components called **`PropTypes`**

Using PropTypes

- Start by importing **`PropTypes`** from the **`prop-types`** module
- For a given component **FooComponent**, define **`FooComponent.PropTypes`** as an object literal
- The keys will be the names of properties
- The values will be type and requirement definitions

-
  ```
  FooComponent.PropTypes = {
    name: PropTypes.string.isRequired,
    age : PropTypes.number
  }
  ```

PropType types

- **`PropTypes.array`**
- **`PropTypes.bool`**
- **`PropTypes.func`**
- **`PropTypes.number`**
- **`PropTypes.object`**
- **`PropTypes.string`**
- **`PropTypes.symbol`**
- Add **`.isRequired`** to require the property in question

More PropTypes

- **`PropTypes.node`**
  - Anything that can be rendered (string, number, HTML element…)
- **`PropTypes.element`**
  - A React element
- **`PropTypes.instanceOf(Car)`**
  - The property must be an instance of a type
- **`PropTypes.oneOf( [ 'foo', 'bar' ] )`**
  - Effectively an enumeration of possibilities

Even more PropTypes

- **`PropTypes.oneOfType( [ PropTypes.string, PropTypes.number ] )`**
  - Must be one of these types
- **`PropTypes.arrayOf( PropTypes.string )`**
  - Should be an array of Strings
- **`PropTypes.objectOf( PropTypes.number )`**
  - Object properties should be of this type
- **`PropTypes.any.isRequired`**
  - Can be any type, but must be present

PropTypes and shape

-
  ```
  PropTypes.shape( {
    name: PropTypes.string,
    age: PropTypes.number
  } )
  ```
- Use **`PropTypes.shape`** to design a custom object
- Or provide for an object that does not have a type or prototype (that is, which cannot use **`PropTypes.instanceOf`**)

Demo: PropTypes in action

- In **sp-react-demos**, under the **demos** folder, look at **PropTypesDemo.js**
- You will see various uses of **`PropTypes`** to ensure that the proper type of data is being passed into a component
- Note that there are several spots where you can comment and uncomment code to see **`PropTypes`** usage fail

Exercise 12: Adding PropTypes

- Objectives
  - Add **`PropTypes`** to **`PayeeDetail`**
  - Add **`PropTypes`** to **`BrowserButtons`**
- Use gulp to load the class files for the exercise
  - **`gulp start-exercise --src ex-12`**
- If you have problems with the exercise, ask your instructor for help

Lists of data

- In previous exercises, we built a view which could page through a list one item at a time
- Realistically, we will also want to be able to render multiple items in a list
  - Think of a table, or just a list of possibilities
- React does not come with any API for rendering lists of data
- Instead, it leverages existing JavaScript functionality

Rendering a list

- A list is more commonly called an array
- JavaScript has multiple tools for iterating over arrays
  - **`forEach`**, **`some`**, **`every`**, **`map`**, **`filter`**, **`find`**, etc.
- As a thought experiment, which tool would be useful here?
- We need to iterate over the elements of the list
- And render them as React components
- This is a sort of transformation of one list into another
  - A list of data into a list of components
- The **`map`** method will be useful here

Using map to render a list

```
// Assumes a list 'people'
{
  people.map( person => {
    return <li>person.firstName person.lastName</li>
  })
}
```

Lists and keys

- Rendering a list can be an expensive operation
  - There are many elements
  - It is difficult to tell whether elements have changes
  - Many changes to the DOM could be required
- React asks for listed elements to add a **key** property
- The **key** property should have a value unique within this list of elements
  - Different lists can have the same key values
- React uses the **key** property to compare items in the list to see if they need re-rendering

Demo: Lists and keys

- In **sp-react-demos**, under the **demos** folder, look at **ListsAndKeys.js**
- Note that there is a custom component for rendering a table row
- We have not attached the key to the **`<tr>`** element, but instead to the **`CustomRow`** component
- Keys should be used at the element or component generated in the loop, not a child or descendant element or component

Exercise 13: Lists and keys

- Objectives
  - Create a component, **`PayeeList`**, which renders a list of Payees into a table
  - This will include a component, **`PayeeRow`**, as well
- Note that some code has been added to **`PayeesComponent`** to manage whether the UI displays **`PayeeList`** or **`PayeeDetail`**
- Also note that when the starter code loads, there will be an error (which will be fixed over the course of the exercise)
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-13`**
- If you have problems with the exercise, ask your instructor for help

Sorting a list

- What are the pieces of the puzzle if we want to sort the list?
- Arrays can be sorted with a native **`sort()`** method
  - Though that method is used for arrays of primitives, not arrays of objects
- Lodash, the JavaScript utility library includes a **`sortBy`** function
  - **`sortBy(someArray, sortingFunction)`**
  - **`sortBy(someArray, [propOne, propTwo])`**
- The **`sortBy`** utility expects to sort arrays of objects
  - It also returns a new array, rather than sorting the array in place
- We would also need an event handler to trigger sorting
  - Probably when we click on the header for a column in the list

Demo: Sorting a list

- In **sp-react-demos**, under the **demos** folder, look at **SortingLists.js**
- We use **`sortBy`** to sort the array of items
- Note that we assign the new array back to the old array value so we can maintain the sorted array
- And clicking on a column header a second time does not reverse the sort (as is customary in many UIs)
- How would you implement a reversed sort?

Exercise 14: Sorting a list

- Objectives
  - Clicking on a column header should sort our list of Payees
  - Clicking on the same column header again should reverse the sort (from ascending to descending and vice versa)
  - When browsing through **`PayeeDetail`**, the browse order should be affected by the current sort
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-14`**
- If you have problems with the exercise, ask your instructor for help

What's next?

- We would like to do three things with our Payees application
  - Search
  - Add a Payee
  - Edit a Payee
- To get these features, we need to acknowledge that React is coming up against some limitations
  - All our state is stored in PayeeContainer, which will not be practical when we try to add or edit a Payee
  - We will be adding three new “views” but have been implementing view management in React, which is not ideal

Adding to the toolkit

- To solve these issues, we will expand our toolkit beyond React
- For state management, we will use the popular Redux library
- For view management, we will add routing to our application with the React Router
- The next few chapters will cover Redux and later React Router in detail
- Then, towards the end of the course, we will return React to work with forms, which we need for our Search, Add, and Edit views

Conclusion

Redux Chapter 6

Introduction to Redux

Chapter preview

- Why separate state management?
- Why Redux?
- About Redux
- Introducing Redux
- Pieces of the puzzle
    - Actions
    - Reducers
    - The store
- Testing Redux

Why separate state management?

- Managing state in React can be complex
  - Particularly as the complexity of an application increases
- Individual components can manage their own state, and some of their children's state
- But components are intended to be UI and view-oriented
- Offloading state management to another tool frees components to focus on their core job
- Additionally, the state manager can be tested independently of the view, which is a significant simplification and benefit

Why Redux?

- There are multiple state management solutions for React
- Redux is probably the most popular (at the moment)
  - Which means more support and examples on the web
- Redux has extensive documentation
- Redux is very fast
- Redux is east to test
- While Redux is not bound to React, bindings to React exist and are well-documented

About Redux

- Website http://redux.js.org
- Maintainer: Dan Abramov (and many others)
- Videos by Dan Abramov:
  - Getting Started with Redux: https://egghead.io/series/getting-started-with-redux
  - Building React Applications with Idiomatic Redux: https://egghead.io/courses/building-react-applications-with-idiomatic-redux
  - Both video courses are free!
- Github: https://github.com/reactjs/redux/

Introducing Redux

- Redux is based on a simple principle: There should be a single source of truth for an application
- This source of truth is represented by a store of data
- Any part of the application can query the store for updated data
- Any part of the application can interact with the store to change data as needed
- Interaction points between Redux and the view are well-defined and clear

Introducing Redux, continued

- Redux is influenced by two libraries
  - Flux for state management
  - Elm for simplicity
- Redux exists as a state tree roughly parallel to React's component tree
- The overlap is not exact
- Think of the human body: React is like the nervous system, with many branches
- Redux is more like a skeleton: the sturdy support structure

The pieces of the puzzle

- Work with Redux is comprised of three types of objects
- The **store**, which stores the current state
- **Reducers**, which manipulate state
- **Actions**, which define what state manipulations are available
- Actions are **dispatched** to reducers which update the store
- This interaction can sometimes feel rigid and overly-specced
- But remember that the goal here is to have a **single source of truth** whose interactions are very clearly defined

Pieces of the puzzle: state

- At the core of Redux's interactions is your application's state
- Redux maintains the state of your application, which is a JavaScript object with various key-value pairs, extending deeply as needed
- This state tree is held by the store
- The interactions with the state tree are defined by actions and executed by reducers

Actions

- Actions define an interaction with a part of the state tree
  - Maybe a branch of the tree, maybe a leaf of the tree, it is up to the interaction
- Actions have a type and other properties
  - The type is usually a string constant like 'ADD_TRANSACTION'
  - The other arguments are the information needed to complete the action
  - In this case, presumably, another transaction
  - Or the fields from which a transaction could be built
- Actions are not required to have other properties
  - Think of a 'CLEAR_SORT' action on a list, for instance
  - But usually there is at least one other property

Action generator

```
const addTransaction = (tx) => {
  return {
    type: 'ADD_TRANSACTION',
    tx: tx
  }
}
```

Action generators

- Having a standalone action object is not very useful
- You would have to customize it for each use of that action
- Use a function to generate the action object
- This kind of function is called an action generator
  - Sometimes an action creator

Demo: Actions

- We will be looking at the same file throughout this chapter
- We will be able to see the interactions of components, a store, reducers, and actions
- The file is in **sp-react-demos** under the **demos** folder as **ReduxCounter.js**
- For actions, look for the section labeled with the comment **`// Actions`**

Exercise 15: Prelude

- Over the next few exercises, we will build a simple example of a Redux-enabled component
- This component is not something we would work with in the real world of React and Redux, but will serve as a prelude to how React and Redux would work together
- It should help us understand better what's going on with Redux and how it works

Exercise 15: Actions

- Objectives:
  - We want to take a Payee and swap it from active to inactive or vice-versa
  - First, we will write an action which will define this interaction
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-15`**
- If you have problems with the exercise, ask your instructor for help

Reducers

- Reducers manipulate state by processing actions
- Reducers are themselves functions which take two arguments:
  - The current state (**or a default**)
  - The action to perform on that state
- Reducers return updated state, depending on the action processed
  - Or the previous state, if no action was processed
- Reducers reduce a state and an action to a new state
  - …and then return that new state
- Reducers are often named after the state element they work with

Reducer function

```
const reducer = (state = {}, action) => {
  switch (action.type) {
    case 'DO_SOMETHING':
      // Update state in one way
    case 'DO_OTHER_THING'
      // Update state in another way
    default:
      return state; // Do nothing
  }
}
```

State manipulation

- Reducers never directly manipulate state
- In fact, Redux state trees should be seen as immutable
- Much like with React, instead of changing state, replace state with new objects
- Reducers should take the current state, copy it, mutate the copy, and then replace the original with the copy
  - Behind the scenes, this makes for rapid comparisons between previous and current states
  - Which allows React to quickly determine what it needs to re-render

Demo: Reducer

- In the **sp-react-demos** project, look under the **demos** folder at **ReduxCounter.js**
- The area marked with a comment **`// Reducer`** is where we can see the reducer defined

Exercise 16: Reducers

- Continuing to work with our rudimentary React-Redux application
- Objectives
  - Add a reducer called **`payee`**
  - It should check to see which **`action.type`** it receives
  - If it is processing **`'TOGGLE_PAYEE_ACTIVE'`** it should modify the state accordingly
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-16`**
- If you have problems with the exercise, ask your instructor for help

Store

- A store is a collection of reducers
- Stores are initialized with state from reducers
  - Which provide default state
- Subsequent interactions with reducers manipulate that state
  - As said, never directly, always through replacements
- This is the first time we actually use a Redux function: **`createStore`**
- Create a store by passing a reducer (or set of reducers) to **`createStore`**, which will return a store

Calling a reducer on a store

- Actions are passed to reducers on a store by a **dispatcher**
- The store has a **`dispatch`** function which takes an **action** as an argument
  - Or a function which returns an action
- Internally, the dispatcher figures out which reducer to call based on the **`action.type`**
- The reducer is called with state (if it exists, default otherwise) and the action object
- The state returned from the reducer is stored internally by the store

Accessing store data

- To see data in the store, there are two options
- At any time, you can call **`store.getState()`** to see the entire state tree of the store
- To find out about store updates, you can invoke **`store.subscribe()`**
- The **`subscribe`** method takes a function as an argument, which will be invoked on any change
- In the listener function, invoke **`store.getState()`** to get the current state of the store
- This is a low-level tool we will not use in later chapters
  - Though it helps us understand what's going on right now

Demo: Redux store

- Continue to look at **ReduxCounter.js** in the **sp-react-demos** project, under the **demos** folder
- Pay attention to the areas labeled
  `// Creating the store`
  and
  `// Working with the store`
  and
  `{/* Dispatching actions */}`

Exercise 17: Stores and tying it together

- Now we will add a store, which will tie together the functionality of our Redux-enabled component
- Objectives
  - Create a store
  - Initialize the state of the component with the state of the store
  - Subscribe to the store with a listener which updates component state
  - Add a button which dispatches an action to the store
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-17`**
- If you have problems with the exercise, ask your instructor for help

Testing Redux

- At the moment, our actions and reducers are locked away in our component file
- We want to test these (of course) but it would be difficult
- Typically, actions and reducers are broken out into separate files which export their respective functions
- This allows for the actions and reducers to be tested independently from the components they work with

Redux tests

- Actual tests are rather uncomplicated
- For action functions, test that the function, given the right input, returns a proper-looking action object
- For reducer functions, passed a state and an action, they should return the correct state
- Store tests are usually redundant if you have tests of actions and reducers

Demo: Redux testing

- We have broken out some of the files from our demo
  - Everything is still in **sp-react-demos** under the **demos** folder though
- The reducer has been moved to **demo-reducer.js**
- The actions have been moved to **demo-actions.js**
- In the **tests** folder under demos, you will find **demo-reducer.spec.js** and **demo-actions.spec.js**, the test files for each of our demos
- Run the tests with **`npm test`** and check out the results

Exercise 18: Writing tests

- We have moved the reducer and action into their own separate files, while keeping the original component functioning
- We have added test files under the **tests** folder for the reducer and action as well
  - The files are empty
- Objectives
  - Write tests for the reducer
  - Write tests for the action
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-18`**
- If you have problems with the exercise, ask your instructor for help

Where do we go from here?

- In this chapter, we worked on a simple example for the sake of understanding concepts
- There are many aspects of Redux we used in this chapter we would not use in the real world
  - Getting state from **`store.getState()`**
  - Using **`store.subscribe()`** directly
  - And so on
- In the next chapter, we will tie React and Redux together "properly" using well-established patterns

Conclusion

Redux Chapter 7

Redux and React

Chapter preview

- Redux and React
- Tools for Redux and React
- Tying state and props together
- Tying events and dispatches together
- Connecting a store to a component
- Testing?

Redux and React

- Redux and React have a lot in common
  - Trees of representations
  - Information flows downward in the tree
  - Or from root to branch to leaf, depending on your perspective
  - Changes trigger re-evaluation
- It is not difficult to envision tying Redux and React together with a small library which would make it easier to manage one from the other
- Which is, in fact, what the **react-redux** module does

Tools for React and Redux

- The react-redux module imports two special tools for working with React under Redux
- The **`Provider`** component makes it easy to make the store available to every branch of the tree
- The **`connect`** function wires together a component and a store with details on how to hook up state to props and dispatch actions based on events
- The module does not really provide much else, for other features we would have to add functionality from Redux or an external library

Providing a store

- The Provider makes the store available to any components under it
- Developers often make Provider the child of App, or vice versa
- Ensuring that the entire application has access to the store
- Pass Provider one argument, store
- The store should be set to an already-configured store
  - Set reducers, apply middleware, dispatch any initializing actions BEFORE you assign the store to the Provider component

Connecting to the store

- The **`connect`** function connects a React component to a Redux store
- It does so through two sets of mappings:
  - state to props
  - dispatch to props
- Conceptually, **`connect`** is plugging the component into Redux at two connect points: props and events
- When the store is updated, it can update your component's props
- When custom events are called, they can dispatch actions to the store, updating the state

The concept of connect

- Think about a simple React component
- It has inputs, which we usually bind as data going from parent to child
- And it has outputs, which are also bound, but act like custom events
- Both are passed through the props property (or this.props in a class-based component)

Using connect with Redux

- Redux manages state
- Redux changes or updates states via dispatched actions
- For your React component, you could argue that Redux's state becomes your component's props
- Similarly, when your component wants to act on that state, it will wind up emitting some kind of event
- Redux should map that event to a dispatch against the store

connect in action

```
let ContainerComponent = connect(
  mapStateToProps,
  mapDispatchToProps
)(ComponentToBeWrapped)
```

Tying state and props together

- The first argument to connect is **`mapStateToProps`**

-
  ```
  mapStateToProps = (state, [ownProps]) =>
            ( return { state to prop map } )
  ```
- **`mapStateToProps`** is a function
- Arguments: store state and, optionally, props passed to the wrapped component
- Returns: An object mapping of store state to props in the wrapped component
- Think of **`mapStateToProps`** as the subscriber to store updates

mapStateToProps

- The mapStateToProps function tells your component how to map Redux changes (state) to the components props
- It is a way of saying, "if this Redux store property changes, I want to update my component with the new value"

Tying events and props together

- The second argument to connect is **`mapDispatchToProps`**
- **`mapDispatchToProps`** can be either a function or an object
- If it is a function…
  - Arguments: The dispatch method from the store
  - Returns: an object mapping of props (usually event handlers) to dispatchers (usually action creators)
- If it is an object, it is what the functional version returns
  - That is, an object mapping of props to dispatchers

mapDispatchToProps

- mapStateToProps covers when the state of Redux updates
- mapDispatchToProps allows us to control how updates work
- Take a custom function on a component, and tie it to a dispatch action
  - This does not have to be the only, or even every-thing that the function does
- Calling "onPayeeUpdate" in a component should wind up calling the "PAYEE_UPDATE" action with the new payee as a payload, for example

Containers and connections

- The Redux docs point out accurately that wrapping a presentational component in a **`connect`** call essentially generates a container component
- The connected component which results is "**smart**" by virtue of **`connect`** plugging the store into the wrapped "**dumb**" component
- The container component is a thin layer which interacts between the presentational component and the Redux store
- This is sometimes called a higher-order component

Demo: A connected component

- In **sp-react-demos**, under the **demos** folder, look at **ConnectedComponent.js**
  - And naviagate to /demos/connected-component to try it out
- Note that this is a variation on the demo component from the last chapter
- But instead of wrapping it "by hand" (as it were), we use connect to plug into the appropriate aspects of the component

Exercise 19: Redux and PayeeDetail

- We are going to rebuild **`PayeeDetail`** (including the **`BrowserButtons`** component) as a component connected to a Redux store
- Objectives:
  - Create a store to manage **`PayeeDetail`**'s state
  - Create action(s) and reducer(s) to interact with the store
  - Connect the store to **`PayeeDetail`** using **`connect`**
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-19`**
- If you have problems with the exercise, ask your instructor for help

Application-level Redux

- In the last exercise, we tied a small-scale component to a Redux store
- But we want to connect the store to many components in our application, or at least be able to
- How can we make a store available to multiple levels of an application?
- The react-redux bindings provide a utility component for this: **`Provider`**

Provider

- The **`Provider`** component should be the root component for your entire application
  - Wrap it around **`App`**, for instance
- Create the store in **`index.js`**, and then pass it as a prop to **`Provider`**
- **`Provider`** takes advantage of a React quasi-global called context
  - We will not be going over context here, as it is not a best practice
  - It may also be going away to be replaced by something else!
- For now, **`Provider`** makes the store available to all its descendant components

Redux-React architecture

- In the last exercise, our container component was defined exclusively by **`connect()`**
- This will not always be the case!
- Container components can exist without being run through **`connect`**
  - Though usually a parent container will then be **`connect`**ed
- Container components can also interact with dispatching directly
  - A container component that has been wrapped with **`connect`** has access to the **`store.dispatch`** method on **`props`** as **`props.dispatch`**

Demo: Redux-React containers

- In **sp-react-demos**, under the **demos** folder, look at **ReactReduxContainers.js**
- You will see some different variations on how to use connect and other Redux tools

Exercise 20: Redux and Payees

- Now we will re-implement Payees, taking advantage of Redux
- Objectives:
  - Create a store to manage **`Payees`** state
  - Create action(s) and reducer(s) to interact with the store
  - Connect the store to **`Payees`** using **`connect`**
  - Wrap the application in **`Provider`**
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-20`**
- If you have problems with the exercise, ask your instructor for help

Testing?

- Should we write tests for the application?
- On the one hand, nothing in the view has changed, so our React-oriented tests should still run fine
- On the other hand, we have changed out state management to Redux, so anything that interacted with state will need to change
- We also have moved our actions and reducers into separate files, for ease of testing
- While the content we are testing has changed, the methodology has not

Exercise 21: Testing our components

- Objectives
  - Go over the React-oriented tests for our components, see if any need changing
  - Write Redux-oriented tests for our actions and our reducers
- Use gulp to load the class files for the exercise
  - **`npx gulp start-exercise --src ex-21`**
- If you have problems with the exercise, ask your instructor for help

Conclusion

React Chapter 8

Thunks, Lifecycle, Routing

Chapter preview

- Middleware
- Asynchronous middleware
- React lifecycle
- Routing
- Forms

Middleware

- Redux allows you to register middleware with your store
- Middleware is code that can interact with the store on dispatches and can also invoke the store's state via getState()
- A basic example of middleware is a logger
- Register a logger as middleware, and the logger will log each dispatched action
- The redux-logger package, for example, registers dispatched actions as well as previous and next states

Registering middleware

- For any given middleware:
- Import applyMiddleware from the redux package
- Call createStore, passing
  - The reducer
  - Optionally, initial state
  - A call to applyMiddleware, passing in middleware code
- Look at the demo on the next slide for details

Demo: Using middleware

- In **sp-react-demos**, under the **demos** folder, look at **ReduxMiddlewareLogger.js**
  - And naviagate to /demos/redux-middleware-logger to try it out
- Note that this is a variation on the ConnectedComponent from the last chapter
- When you examine the demo have the console open in the developer tools of your browser
- Note how the logger outputs information each time you increment or decrement the state

Demo: Using middleware

```
import { createStore, applyMiddleware } from 'redux';
import logger from 'redux-logger';

// Other code here

// Store
const store = createStore( reducer, applyMiddleware(logger) );
```

Redux devtools

- The Redux logger is nice, but there is a better set of development tools for Redux: the Redux devtools
- The Redux devtools are a combination of middleware and a browser plugin which displays a variety of useful, well-organized information about your Redux store
  - If, for some reason, you cannot use the browser plugin, you can use the dev tools in a web page as well
- The home page for the Redux devtools is https://github.com/zalmoxisus/redux-devtools-extension
- That page has directions for installing the browser plugin

Configuring Redux devtools

- The devtools, like the logger, are added on as middleware for your Redux store
- The configuration differs if you are using the devtools as the only middleware, or as one of several pieces of middleware code
- If the devtools are the only middleware you are registering, then configure your middleware as follows:

```
// Store
const store = createStore( reducer,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
    window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

Demo: Redux devtools

- In **sp-react-demos**, under the **demos** folder, look at **ReduxDevtools.js**
  - And naviagate to /demos/redux-devtools to try it out
- If you have not done so already, please install the Redux devtools extension for your browser
- When you bring up the demo, open the developer tools in your browser and go to the Redux tab
- Use the incrementer and decrementer buttons as normal, watching what happens in the devtools

Configuring devtools with other middleware

- When configuring the devtools with other middleware, the code is somewhat more complicated
- The example at the home page for the devtools is sufficient for most cases
- When we add more middleware shortly (for asynchronous stores), we will return to configuring the devtools as part of a set of middleware

Asynchronous Redux

- Using Redux asynchronously introduces a new set of complications
- Chief among these is how to deal with an asynchronous process
- An async request goes through two of three possible phases
  - First the request is sent
  - Then, either the request completes successfully
  - Or it fails
- Use Redux, these are three different states:
  - Loading
  - Loading complete & success
  - Loading complete & failure

Asynchronous actions

- There are several different patterns for invoking actions asynchronously
- We will cover a low-level, simple-but-effective pattern, using a concept called a thunk
- The thunk will allow us to tell Redux to run a series of branching actions
    - Which matches the series of request -> response or error in general HTTP requests

What's a thunk?

- A thunk is a subroutine created to assist a call to another subroutine
- In functional programming, thunks are used to pass work or behavior from one function to another
- Because functions are first-class citizens in JavaScript, thunks are a logical way to pass executable code from one part of our application to another
- When we write a thunk, it will be a function which returns a function to be executed later

Redux and thunks

- Redux does not handle a thunk natively
  - Remember that Redux stores expect action objects, not functions
- Adding the redux-thunk package as middleware allows Redux to handle thunks
  - Technically, the middleware handles the thunk and passes proper action objects to Redux
- Register the thunk middleware with your Redux store
- Invoke action generators which return functions the same way you would action generators which create objects

Setting up for async

- Import thunk from redux-thunk
- Import applyMiddleware from redux
- When creating the store, use middleware
  - createStore(reducers, applyMiddleware(thunk))
- And you are done!
- From now on, your action creators can return functions which will run code
  - The returned functions should, in turn, dispatch actions
  - They can still return just objects, too

Demo: Thunks, middleware, and async

- In the sp-react-demos folder, look at three demos:
  - ReactNoReduxNoThunk.js : A starter with hard-coded values and no asynchronous actions
  - ReactNoReduxFetch.js: No Redux, but asynchronous behavior
  - ReactReduxFetch.js: Now move the state manipulation into Redux
- The last demo is a simple system which uses Redux and a thunk to asynchronously return data
  - Note that you should execute **`npm run rest`** (in the sp-react-class project, NOT sp-react-demos) before looking at the last demo
  - This runs a RESTful server, which the Redux store will talk to

Thunk code

- Let's take a look at a thunk
- In ReactReduxFetch.js, there is a method, peopleFetchData which returns a function
- This returned function is the thunk
- It expects to be passed the store's dispatcher (the redux-thunk middleware handles this for you)
- When executed, it dispatches a notification that it's loading (a request is in progress)
- And then will resolve with either data or an error, dispatching different actions in either case

Lifecycle

- When should I execute my asynchronous calls to populate data?
- In the last demo, you may have noticed that PersonList had a function, componentDidMount, where the fetch call was executed
- React components have an extensive, useful lifecycle which we can hook in to according to our needs
- Lifecycle methods are only available for overriding in class-based Components

Lifecycle methods: The DOM

- constructor(props): Obvious
- static getDerivedStateFromProps: is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.
- render(): Obvious
- componentDidMount(): Runs after the component mounted

Lifecycle methods: Updates

- shouldComponentUpdate(nextProps, nextState): A hook to determine whether this component should re-render
  - Defaults to returning true (re-rendering on every state change)
  - Be careful about updating, or more accurately, thoughtful
  - Not called on initial render()
- getSnapshotBeforeUpdate(prevProps, prevState): invoked right before the most recently rendered output is committed to e.g. the DOM.
- componentDidUpdate(prevProps, prevState): Called immediately after an update; not called on initial render(); not called if shouldComponentUpdate() returns false

Routing

- Routing is the concept of changing views in response to changes to the URL
  - Normally, changing the URL would reload the application
  - Routing libraries intercept changes to the URL and update the view accordingly
- Routing has many advantages
  - Changes in view are now entries in browser history
  - Browser forward and backward buttons work within application state
  - Urls can be used to bring up particular views (e.g., /tx/detail/5 could immediately load the detail view for transaction number 5)

Routing and React

- There are several routing packages for React
- We will use the most popular one: react-router
- URL: https://reacttraining.com/react-router/
- Docs: https://reacttraining.com/react-router/web/guides/quick-start
- GitHub: https://github.com/ReactTraining/react-router

Routing concepts

- Import react-router
- Wrap the application in a <Router> component
  - If you are using Redux, the Redux <Provider> wraps react-router's <Router>

  -
    ```
    <Provider store={store}>
      <Router>
        <!-- Application components-->
      </Router>
    </Provider>
    ```
  - The <Router> component can only have one child element
    - Though there can be much complexity under that one element

Basic routing

- Use individual <Route> elements to specify a path and a component to load if that path is activated
- <Route path="/list" component={ListComponent}/>