505 React

Meetup 3

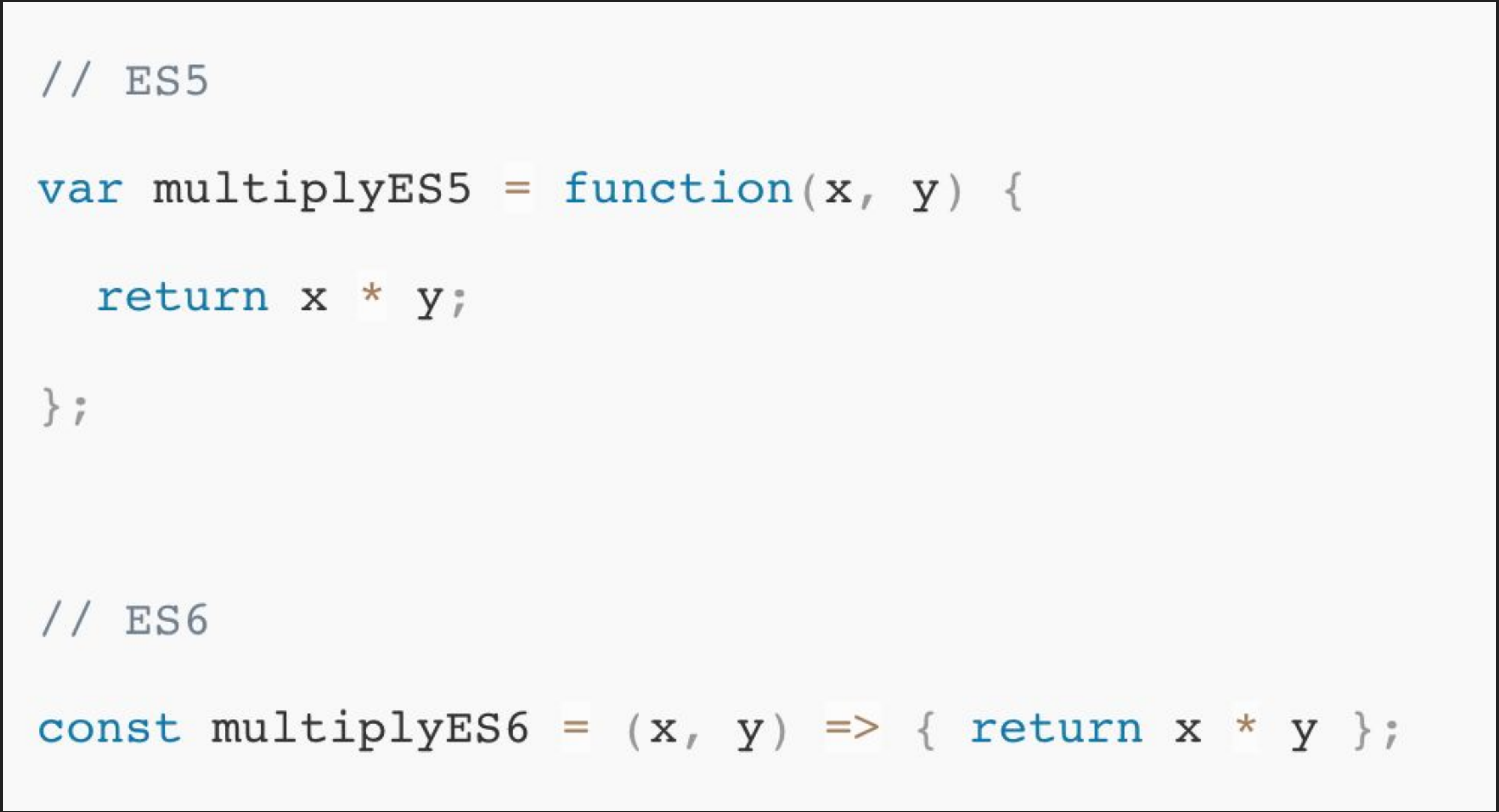
Before we begin

Make sure you have the code from our last meet up and have followed all the set up directions in the slides from the last meetup

https://github.com/samanthaandrews/505-React-Meetup

Arrow Functions

- More concise syntax
- Simplify function scoping and the `this` keyword
- Work much like lambdas in C# or Python
- We don’t have to use the `function` keyword, the `return` keyword, or curly brackets



```
// ES5

var multiplyES5 = function(x, y) {

  return x * y;

};


// ES6

const multiplyES6 = (x, y) => { return x * y };
```

Curly brackets aren’t required if only one expression is present; return is implicit

```javascript
const multiplyES6 = (x, y) => x * y;
```

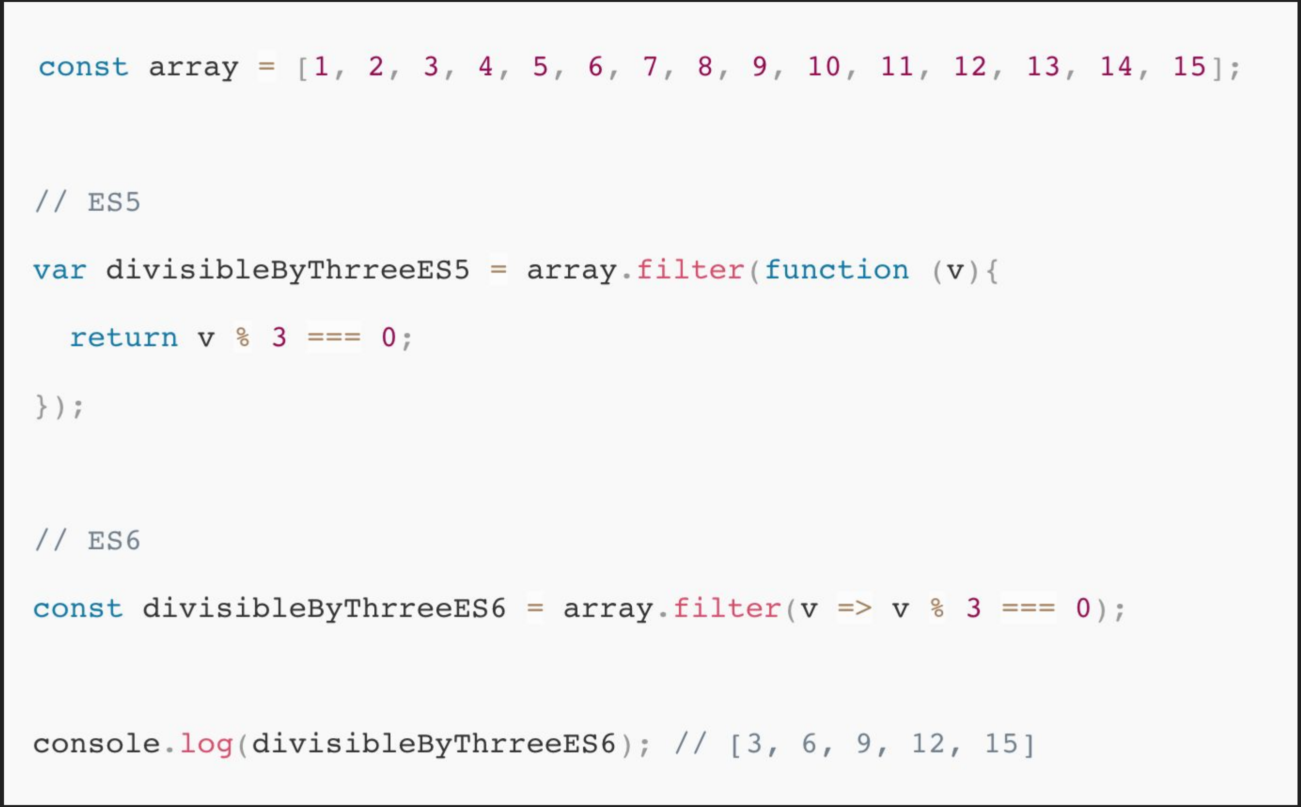
Parentheses are optional when only one parameter is passed
Use empty parentheses when passing 0 parameters

```javascript
const phraseSplitterEs6 = phrase => phrase.split(" ");
```

Wrap object literals in parentheses

```javascript
var func = () => ({foo: 1});
```

Common Use Case for Arrow Functions: Array Manipulation



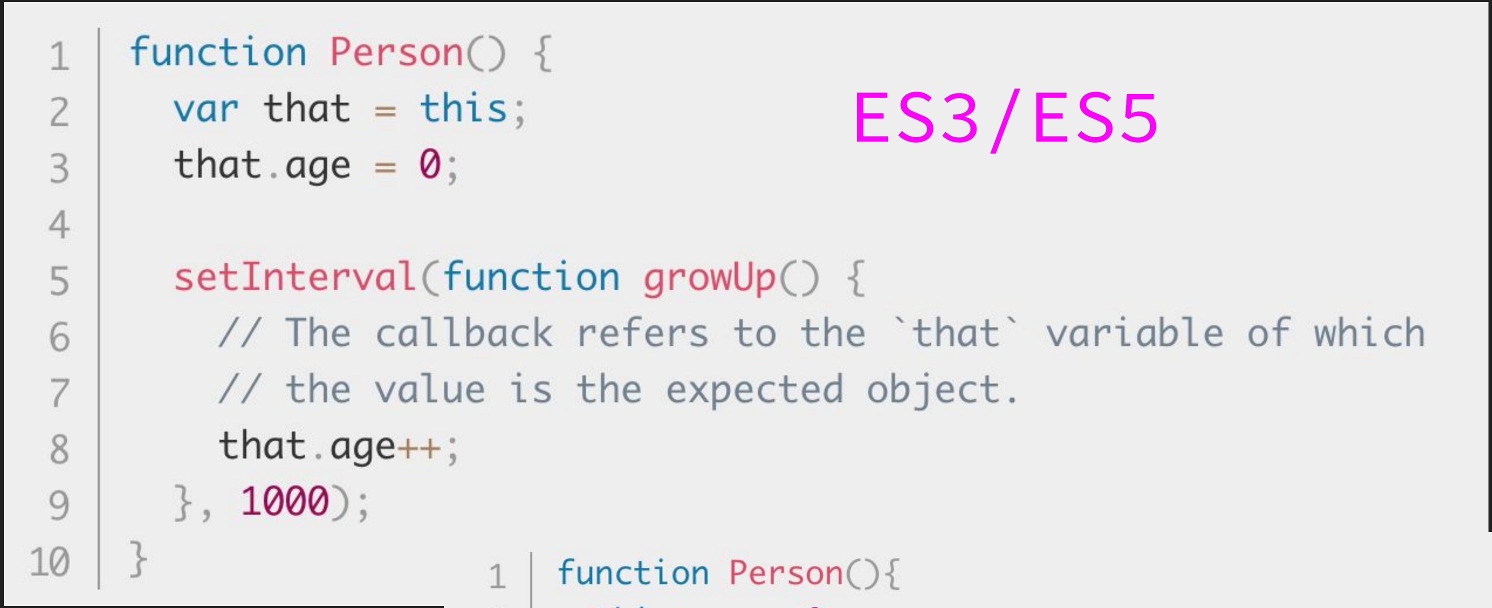
```
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];


// ES5
var divisibleByThrreeES5 = array.filter(function (v){
  return v % 3 === 0;
});


// ES6
const divisibleByThrreeES6 = array.filter(v => v % 3 === 0);


console.log(divisibleByThrreeES6); // [3, 6, 9, 12, 15]
```

Let’s talk about `this`

Until arrow functions, every new function defined its own `this` value (based on how function was called, a new object in the case of a constructor, undefined in strict mode function calls, etc.). This proved to be less than ideal with an object-oriented style of programming.

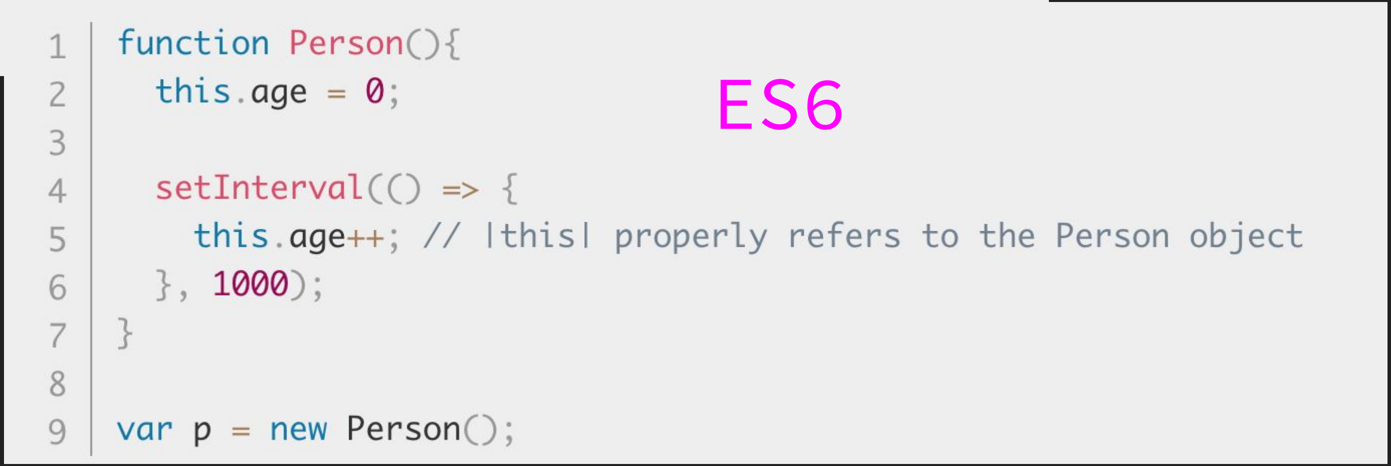
An arrow function does not have its own `this`, `this` comes from the surrounding lexical content.

i.e. arrow functions follow normal variable lookup rules



ES3/ES5

```
function Person() {
  var that = this;
  that.age = 0;

  setInterval(function growUp() {
    // The callback refers to the `that` variable of which
    // the value is the expected object.
    that.age++;
  }, 1000);
}
```



ES6

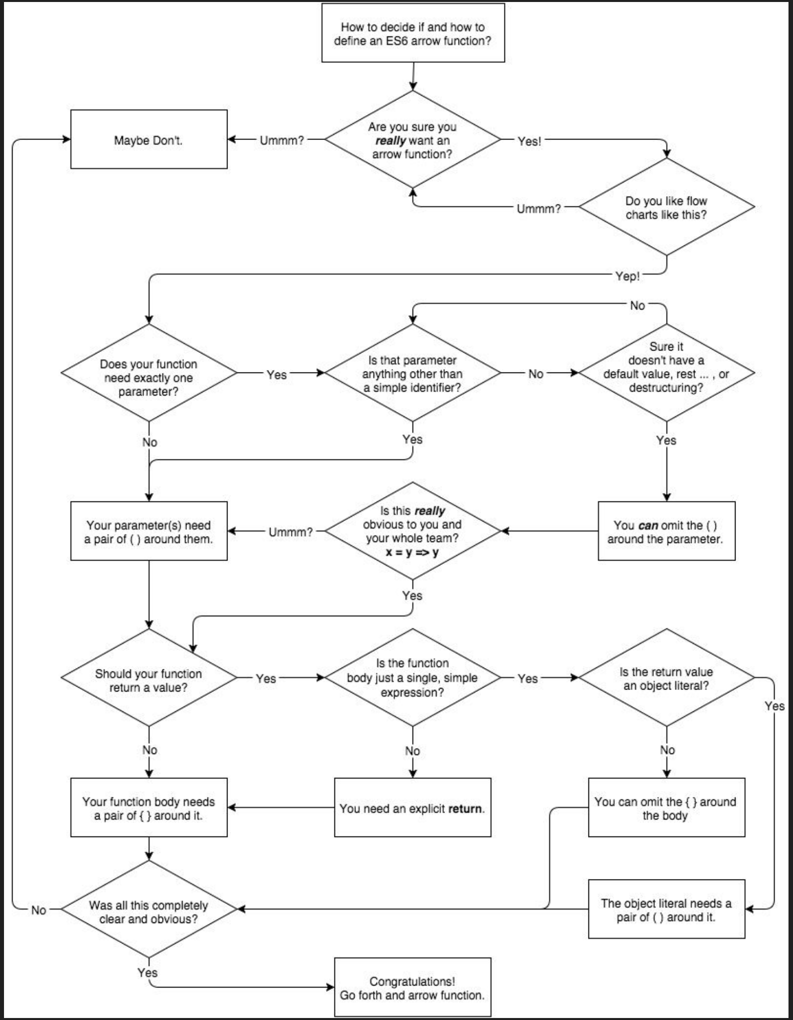
```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the Person object
  }, 1000);
}

var p = new Person();
```

Arrow Function Pitfalls

- Cannot be used as constructors.
- Do not have their own `arguments` object. Best to use rest parameters instead.
- Since arrow functions do not have their own `this`, the methods `call()` or `apply()` can only pass in parameters. `thisArg` is ignored.
- Best suited for non-method functions.
- Arrow functions do not have a `prototype` property.
- Using the `yield` keyword in ES6 arrow functions will throw an error. Use ES6 generators instead.

Arrow Function Pitfalls - Arrow Functions Used As Methods Example

```javascript
'use strict';

var obj = {
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
    console.log(this.i, this);
  }
}

obj.b(); // prints undefined, Window {...} (or the global object)
obj.c(); // prints 10, Object {...}
```


How to decide if and how to define an ES6 arrow function?
Are you sure you *really* want an arrow function?
Ummm?
Maybe Don't.
Yes!
Do you like flow charts like this?
Ummm?
Yep!
Does your function need exactly one parameter?
Yes
Is that parameter anything other than a simple identifier?
No
Sure it doesn't have a default value, rest ... , or destructuring?
No
No
Yes
Yes
Your parameter(s) need a pair of ( ) around them.
Ummm?
Is this *really* obvious to you and your whole team? **x = y => y**
You *can* omit the ( ) around the parameter.
Yes
Should your function return a value?
Yes
Is the function body just a single, simple expression?
Yes
Is the return value an object literal?
Yes
No
No
No
Your function body needs a pair of { } around it.
You need an explicit **return**.
You can omit the { } around the body
The object literal needs a pair of ( ) around it.
No
Was all this completely clear and obvious?
Yes
Congratulations! Go forth and arrow function.

How to decide to use an ES6 arrow function

Flow chart by Kyle Simpson from the book *You Don't Know JS: ES6 & Beyond*

When should I use arrow functions?

According to developer Lars Schoning

- Use `function` in the global scope and for `Object.prototype` properties
- Use `class` for object constructors
- Use `=>` everywhere else

Kevin Smith (a random guy) counted function expressions in various popular libraries/frameworks and found that roughly 55% of function expressions would be candidates for arrow functions.

Redux

- Redux is a predictable state container for JavaScript apps. It is an open-source, JS library.
- No matter the size of your application, the whole state of your app is stored in an object tree inside a single *store*.
- The only way to change the state tree is to emit an *action*, an object describing what happened.
- To specify how the actions transform the state tree, you write pure *reducers*.

Why Use Redux?

- Our code must manage more state than ever before. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. Plus UI state!?
- Managing this ever-changing state is hard. At some point, you no longer understand what happens in your app as you have **lost control of the when, why, and how of its state.**
- Redux attempts to make state mutations predictable by imposing certain restrictions on how and when updates can happen.

When it makes sense to use Redux

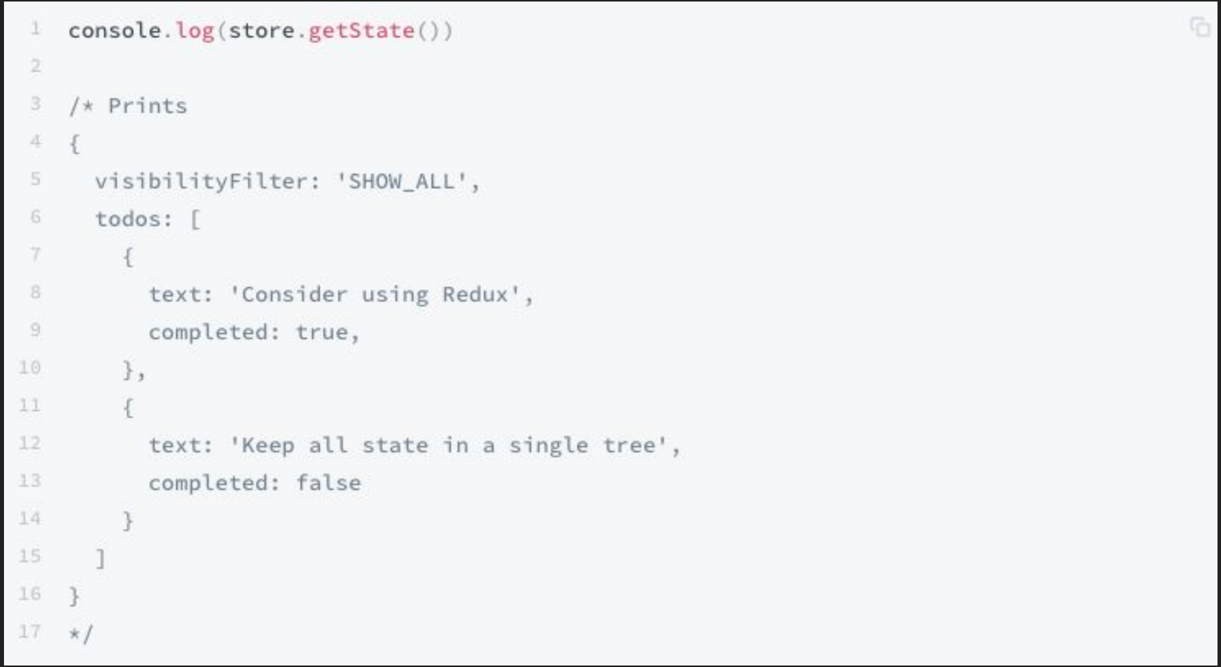
- You have reasonable amounts of data changing over time
- You need a single source of truth for your state
- You find that keeping all your state in a top-level component is no longer sufficient
- You don’t want to pass props down various generations

Three Principles of Redux

1. **Single source of truth** - the state of your whole application is stored in an object tree with a single store.
2. **State is read-only** - the only way to change the state is to emit an action, an object describing what happened.
3. **Changes are made with pure functions** - to specify how the state tree is transformed by actions, you write pure reducers.

Three Principles of Redux

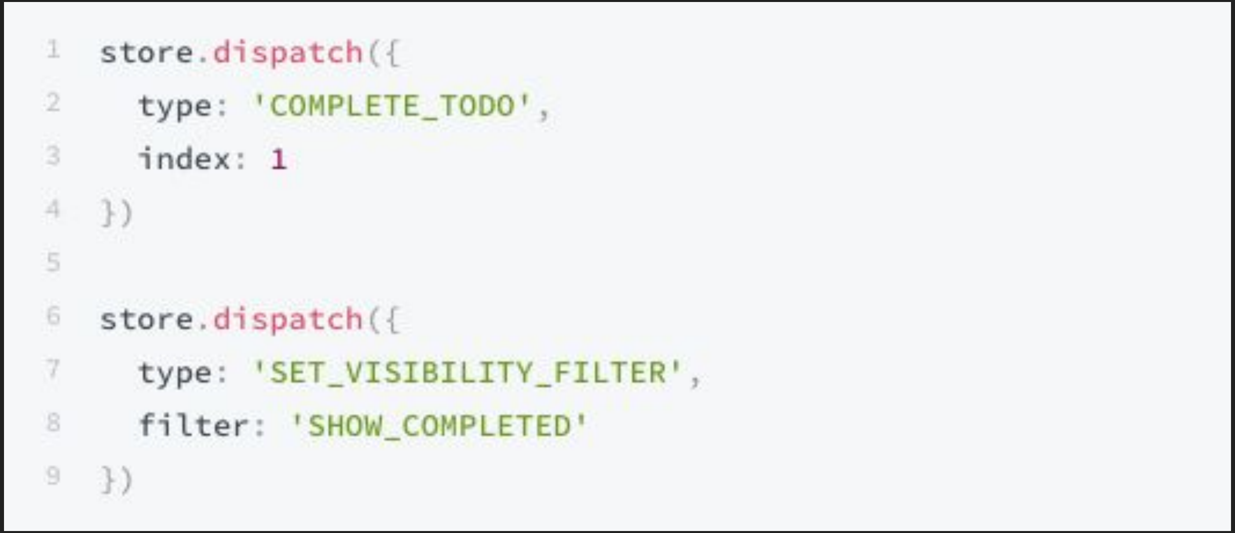
1. **Single source of truth** - the state of your whole application is stored in an object tree with a single store.



```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

Three Principles of Redux

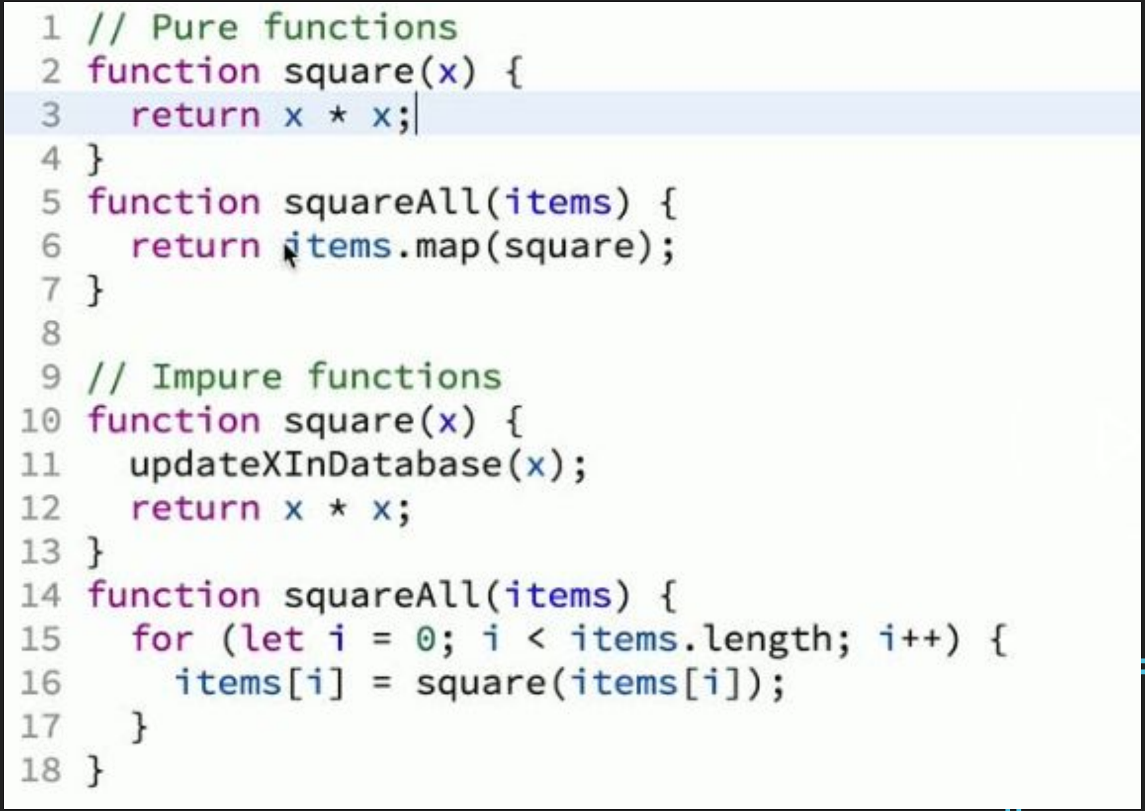
2. State is read-only - the only way to change the state is to emit an action, an object describing what happened.



```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

Three Principles of Redux

The third principle is dependent upon an understanding of pure vs. impure functions

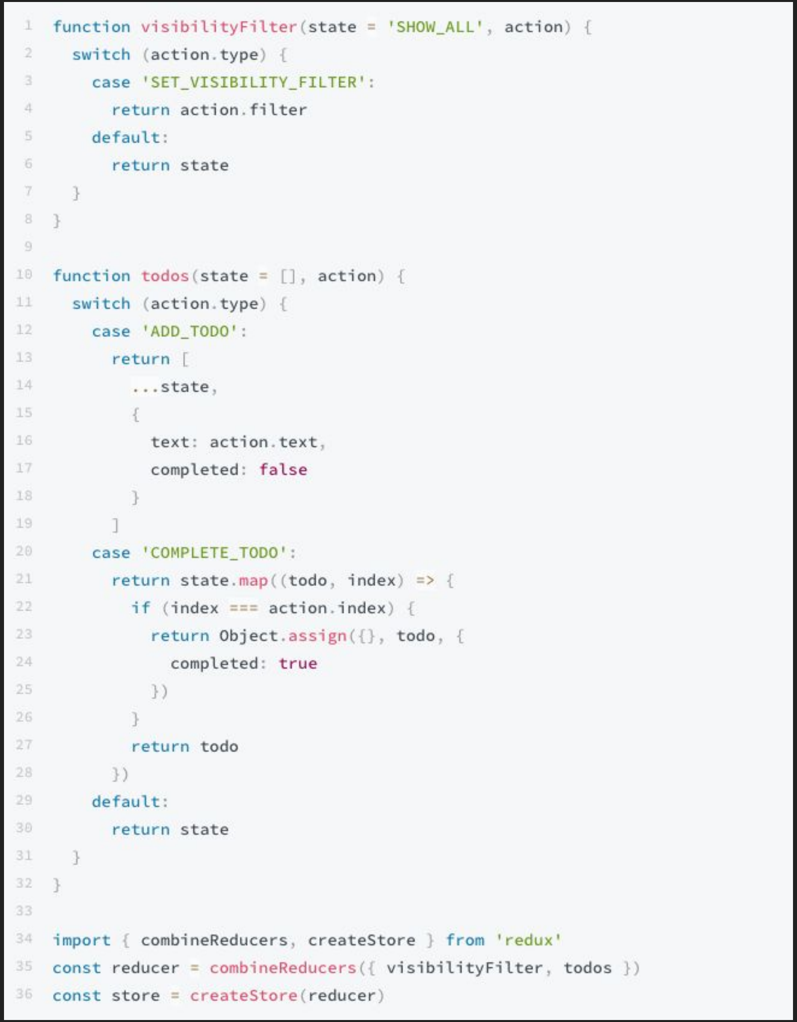


```
// Pure functions
function square(x) {
  return x * x;
}
function squareAll(items) {
  return items.map(square);
}

// Impure functions
function square(x) {
  updateXInDatabase(x);
  return x * x;
}
function squareAll(items) {
  for (let i = 0; i < items.length; i++) {
    items[i] = square(items[i]);
  }
}
```

Three Principles of Redux

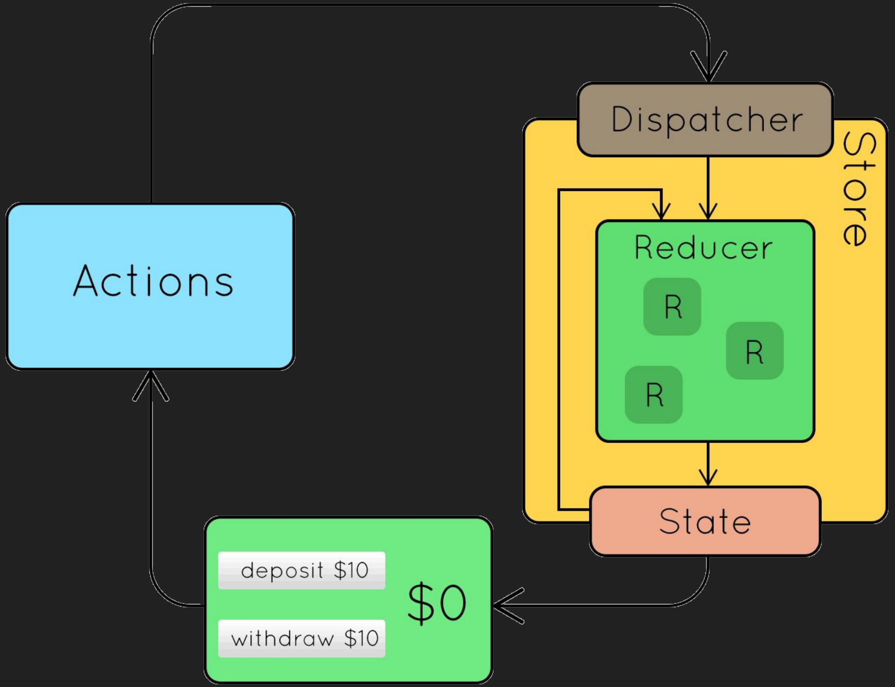
3. Changes are made with pure functions - to specify how the state tree is transformed by actions, you write pure reducers.

A reducer is just a function that takes the previous state & the action being dispatched, and returns the next state of your application.



```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}

import { combineReducers, createStore } from 'redux'
const reducer = combineReducers({ visibilityFilter, todos })
const store = createStore(reducer)
```

Redux


Dispatcher
Store
Reducer
R
R
R
Actions
State
deposit $10
withdraw $10
$0

In case you don’t have it installed

Install Node.js for your operating system

https://nodejs.org/en/

In terminal run the following command

```
sudo npm i -g create-react-app@1.4.1
```

Enter your password

If you get an error, it may be because you installed Node via Homebrew. We recommend uninstalling node and reinstalling directly from https://nodejs.org/en/

In terminal run the following command

```
create-react-app
/Users/[username]/Desktop/trivia-app
--scripts-version=1.0.14
```

This will create a new directory, named “chat-app”
on your Desktop that will hold our code

In terminal run the following commands

```
cd /Users/[username]/trivia-app
```

This will get you into your new React app folder

```
npm start
```

This will launch your new app in the browser

In terminal run the following commands

```
npm install --save react-redux
npm install --save redux
```

This will install react-redux and redux in your project

Add the Provider to your app

This will “provide” the store to the child components

Create the store

This is what will “hold” your state

Create the store

This is what will “hold” your state

Create the actions

This is what defines the functions that modify the store.

You can more than one document outlining actions for cleanliness if you want.

Create the a reducer

This is what controls and changes your state in the store.

You can have more than one per project so we combine them just in case

Connect a component

This will make the store and/or the actions available for you to use

Resources we love:

- https://www.fullstackreact.com/
- https://github.com/getify/You-Dont-Know-JS