Modern JavaScript

The Pros and Const

ES5 is garbage

In My Opinion

ES6+ Is actually amazing

Basic Language Features

Object and Array destructuring

```
var obj = { hey: 'hello', bye: 'ciao' };

var hey = obj.hey;
var bye = obj.bye;
```

```
const obj = { hey: 'hello', bye: 'ciao' };

const { hey, bye } = obj;
```

```
var array = [1, 2, 3, 4];

var first = array[0];
var second = array[1];

var rest = array.slice(2);
```

```
const array = [1, 2, 3, 4];

const [first, second, ...rest] = array;
```

Arrow Functions

```
var foo = function() {
  this.x = 'hello';
};

foo.prototype.bar = function() {
  setTimeout(function() {
    console.log(this.x.length);
  }, 1000);
}
```

```
const foo = function() {
  this.x = 'hello';
};

foo.prototype.bar = function() {
  setTimeout(
    () => console.log(this.x.length)
  , 1000);
}
```

Arrow Functions bind this to the “lexical” this.
Normal functions bind to the caller. For example, the bar function has “this” equal to the foo instance that called the function.

Classes

```
var Base = function() {
    this.baseProp = 42;
};

Base.prototype.baseMethod = function() {
    console.log(this.baseProp);
};

Base.foo = function() {
    console.log("This is foo");
};
```

```
class Base {
    constructor() {
        this.baseProp = 42;
    }

    baseMethod() {
        console.log(this.baseProp);
    }
    static foo() {
        console.log("This is foo");
    }
}
```

Classes

```
var Derived = function() {
    Base.call(this);
    this.derivedProp = "the answer";
};
Derived.prototype = Object.create(Base.prototype);
Derived.prototype.constructor = Derived;

Derived.prototype.baseMethod = function() {
    Base.prototype.baseMethod.call(this);

    console.log(this.derivedProp);
};

Derived.prototype.derivedMethod = function() {
    this.baseMethod();
    console.log(this.derivedProp);
};
```

```
class Derived extends Base {
    constructor() {
        super();
        this.derivedProp = "the answer";
    }

    baseMethod() {
        super.baseMethod();
        console.log("new stuff");
    }

    derivedMethod() {
        this.baseMethod();
        console.log(this.derivedProp);
    }
}
```

let/const

```
var i = 0
if (true) {
  var i = 1;
}
console.log(i); // 1
```

```
let i = 0;
if (true) {
  let i = 1;
}
console.log(i); // 0
```

```
const hey = 'hey';

hey = 'bye'; // TypeError: Assignment to constant variable
```

Almost no reason to use var anymore.

But be aware that const objects can still be mutated, just not reassigned

More Advanced Language Features

Default arguments

```
var initialState = '';
function reducer(state, action) {
  if (state === undefined) {
    state = initialState;
  }
  ...
}
```

```
function reducer(state = '', action) {
  ...
}
```

Destructuring Arguments

```
function Component(props) {
  var message = props.message;

  return (
    <Text>{message}</Text>
  );
}
```

```
const Component = ({ message }) => (<Text>{message}</Text>);
```

Object Shorthand Syntax

```
// some redux reducers
var user = function() ...;
var businesses = function ...;


var reducer = combineReducers({
  user: user,
  businesses: businesses,
};
```

```
// some redux reducers
const user = () => ...;
const businesses = () => ...;

const reducer = combineReducers({ user, businesses });
```

Immutability Shorthand

(not actually in ES6, but is supposed to go into the standard published in 2018, and most Babel presets already include it)

```
var obj1 = { hey: 'hey', foo: 'bar', baz: 'blah' };

// add bar: foo and change baz to equal baz

var obj2 = Object.assign(
  {},
  obj1,
  { bar: 'foo', baz: 'baz' },
);
```

```
const obj1 = { hey: 'hey', foo: 'bar', baz: 'blah' };

// add bar: foo and change baz to equal baz

const obj2 = {
  ...obj1,  // object spread operator
  bar: 'foo',
  baz: 'baz',
};
```

Promises -- Async/Await

```
function businessSearch(lat, lng, radius) {
  return businessSearchApi(lat, lng, radius)
    .then(function(response) {
       return response.json();
    })
    .then(function(response) {
       return transformResponse(response);
    })
    .catch(function(err) {
      return [];
    });
}

businessSearch(lat, lng, radius).then(function(businesses) {
  updateUi(businesses);
});
```

```
const businessSearch = async (lat, lng, radius) => {
  try {
    const response = await businessSearchApi(lat, lng, radius);
    const jsonResponse = await response.json();

    const businesses = transformResponse(jsonResponse);
    return businesses;
  } catch (err) {
    return [];
  }
}

// async functions wrap return values in a promise
businessSearch(lat, lng, radius).then(function(businesses) {
  updateUi(businesses);
});
```

Functional Programming

An extremely shallow overview

I’ve sorta talked about this before!

Reactive Programming (Rx) borrows a lot of concepts from Functional Programing

Definition

functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data

Wat?

Pure Functions

Pure Functions

Have no side-effects

- Does not depend on anything other than the arguments to the function
  - Does not read or set any global variables
- Idempotent - f(x) => y
  - Argument x will ALWAYS evaluate to y
- Does not perform I/O
- Does not mutate its arguments

Performance

- Can be run in parallel without any locking
- Computed value can be cached
  - Memoization

Pure Functions In Practice

Redux

- Your reducers MUST be pure functions
- const reducer = (previousState, action) => newState
  - Given the same previous state, and same action, the computed newState will always be the same
  - Predictability is Priceless

Unit Testing

- Pure Functions are ∞ easier to test
- Even if you are not doing functional programming, thinking about pure functions will help with testing

Side-Effects

- Applications WILL have side-effects
  - We have to do API calls, file I/O, etc
- Quarantine your side-effects

First Class Functions and Higher Order Functions

- Functions are considered to be first-class citizens
- Can be passed around as arguments or return values just like regular variables

```
const createAdder = x => {
  return function(y) {
    return x + y;
  };
};
const oneAdder = createAdder(1);
oneAdder(2); // 3
```

```
const array = [1, 2, 3, 4, 5, 6];
const doubledEven = array
  .filter(x => x % 2 == 0)
  .map(x => x * 2);
```

```
class MapScreen extends React.Component {
  // blah blah blah
}

const mapStateToProps = state => ({
  selectedBusiness: state.businesses.selected,
});

const mapDispatchToProps = dispatch => ({
  fetchBusinesses(lat, lng, radius) {
     dispatch(businessSearch(lat, lng, radius));
  }
});

export default connect(mapStateToProps, mapDispatchToProps)(MapScreen);

/* connect returns a function
   that function takes a component as an argument and returns another component
*/
```

Refactoring Time!

```
let response = await fetch(url);
respBody = await response.json();
let newBusinesses = {};
for (let i = 0; i < respBody.length; i++) {
  let business = respBody[i];
  if (business.promotions.length > 0) {
    newbusinesses[business.uid] = augmentBusiness(business);
  }
}
```

```
let response = await fetch(url);
respBody = await response.json();

let newBusinesses = respBody
  .filter(business => business.promotions.length > 0)
  .map(augmentBusiness)
  .reduce((acc, business) => {
    acc[business.uid] = business;
    return acc;
   }, {});


const reduced = [1, 2, 3].reduce((acc, number) => acc + number, 0);
// reduced = 6
```

Currying

Definition

In mathematics and computer science, currying is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument.

Wat?

```
function something(arg1, arg2, arg3, arg4) {
  // do stuff with arguments
}
```

```
const something = arg1 => arg2 => arg3 => arg4 => {
  // do stuff with arguments
};

const result = something(1)(2)(3)(4);
```

The adder function I showed earlier was a curried function, and the React-Redux connect function relies on the concept of currying.

```
const fetchBusinesses = (lat, lng, radius) => async (dispatch, getState) => {
  const businesses = await businessSearchApi(lat, lng, radius);
  const viewModels = businesses.map(business => {
    return { name: business.short_name, ...etc };
  });
  dispatch(displayBusinesses(viewModels));
};
```

Thunks are dispatched as if they were regular Redux actions

- Redux actions are normally JavaScript objects

dispatch(fetchBusinesses(lat,lng,radius))

Redux-thunk is responsible for the second function call, and it will inject dispatch and getState for you

```
// unit tests
describe('fetch business thunk', () => {
  it('dispatches viewmodels', () => {
    // set up spys for api

    // we inject our own dispatch function, which is a mock
    const dispatch = jest.fn();
    const result = fetchBusiness(lat, lng, radius)(dispatch);

    const expectedArguments = ...;
    expect(dispatch).toBeCalledWith(expectedArugments);
  });
});
```

In our unit test, we give our own implementation of dispatch, that does nothing, but we can very easily inspect it's arguments.

If fetchBusinesses relied on the current state, we could provide mock implementations of getState

Immutability

- React and Redux rely on immutability in order to do quick shallow comparisons
  - Checking references is super quick
  - Checking equality of each property is slow
  - If you mutate data in Redux, you might see weirdness because Redux assumes nothing has actually changed
- Map/Filter/Reduce/etc return new instances of Arrays and Objects
- If this sounds inefficient... you're right
  - But it's not that bad
  - More than make up for it in simplicity and how much faster shallow comparisons are

Recap

- Pure functions force you to not mutate state changes, and reduce side-effects
  - Easier to test
  - Performance gains with parallelization, caching and compiler optimization
- Higher Order functions allow you to pass and return functions
  - Easier to read, more like a recipe
  - Abstract iteration away
- Currying
  - Take arguments one at a time
  - Enables frameworks to do easy dependency injection
- Immutability
  - React and Redux are very focused on immutability

Fin.