Modern JavaScript

Manulife

Agenda

What is Javascript?

- Brief Overview

Why modern JavaScript

- Because ...
- Versatile
- History
- Exercise

Characteristics of JavaScript

- Important aspects
- Important aspects of ES6
- Common concepts
- ES6 features
- Exercise

What is JavaScript?

What is JavaScript?

Brief Overview

- High-level, interpreted programming language
  - A lot of abstraction, unlike low-level languages
  - Executed without needing to run through a compiler
- Conforms to the **ECMAScript** specification (ES6 / ES2015)
- Can run in both browser / client, as well as server side (Node.js)

Why modern JavaScript?

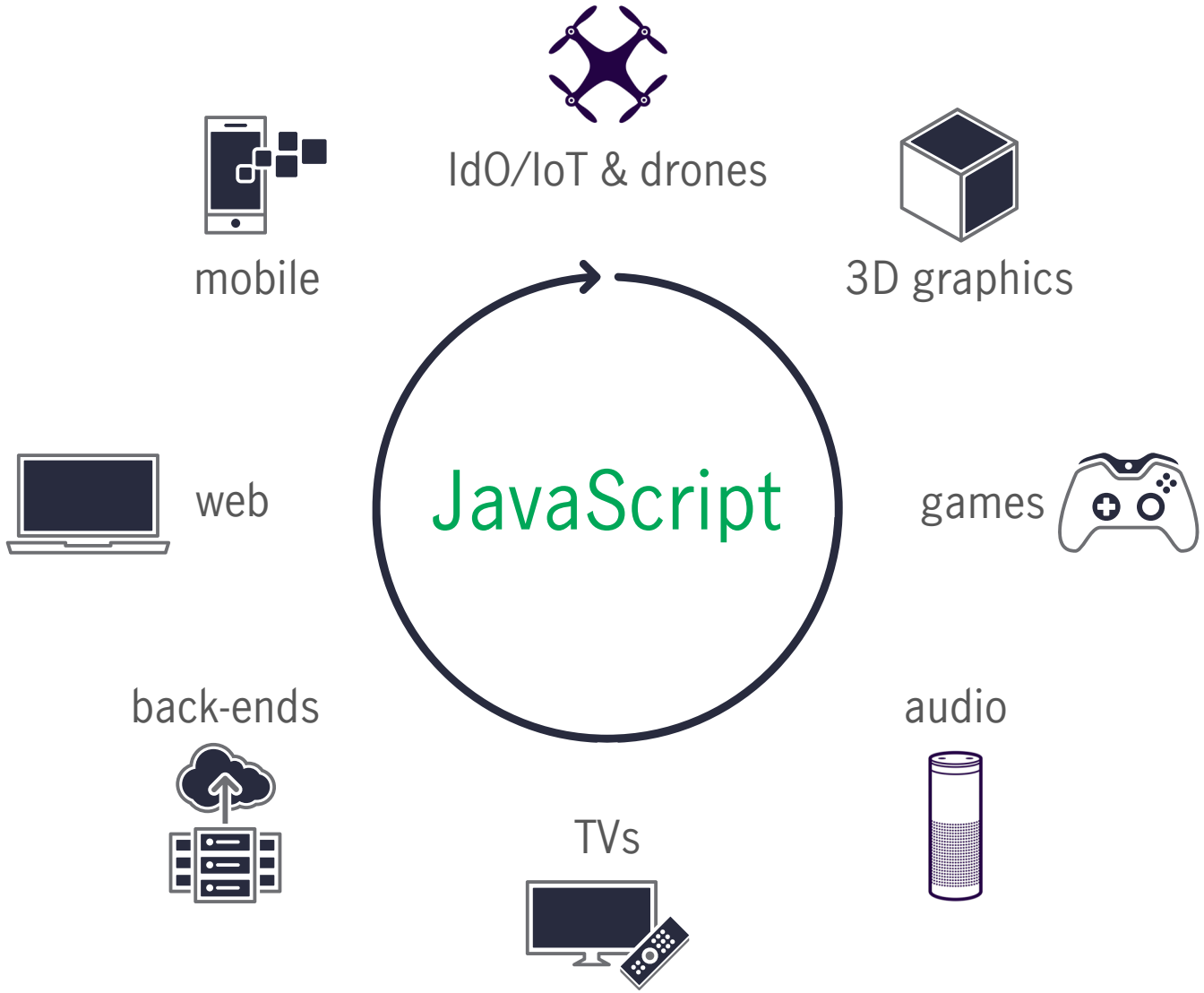
Why modern JavaScript?

Because...

- It's a continuously evolving full-stack platform with cross-browser support
- The JavaScript ecosystem is everywhere around us
- It's a core element of many modern technologies
  - Ex. **React** and **GraphQL**
- It supports multiple paradigms
  - Imperative
  - Functional
  - Object oriented (OOP)

Why modern JavaScript?

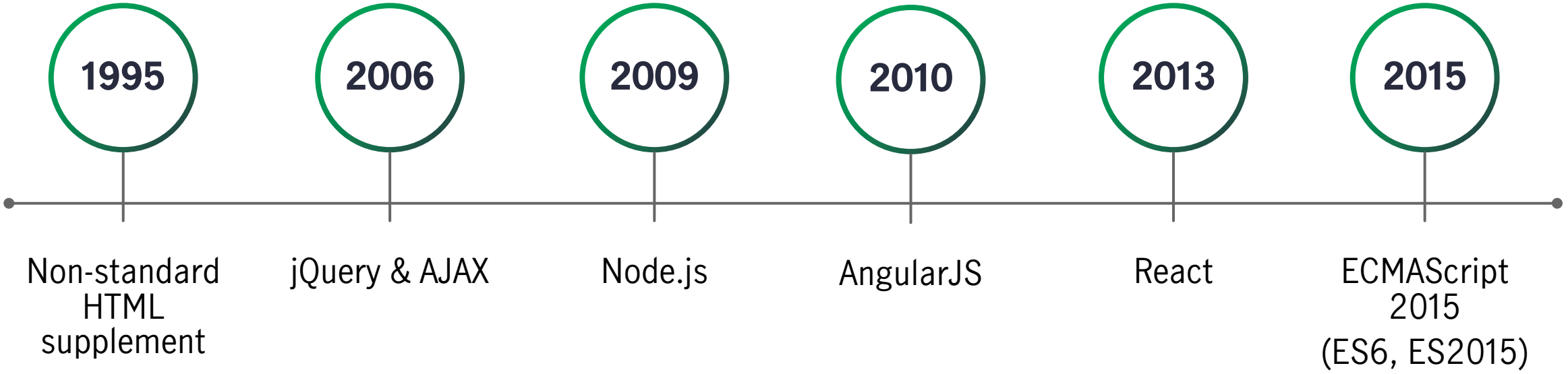
Versatile



Manulife

Why modern JavaScript? : History

- JavaScript has evolved into a functional programming language as well as OOP



Characteristics of JavaScript

Characteristics of JavaScript : Important aspects

1 *Functions as first class citizen*

Functions can...

- Be the only thing in a file
- Be mapped to a dictionary key
- Have properties and methods representing a class

2 *Higher order functions*

- Functions that receive or return a function
- There are many uses throughout the language

Characteristics of JavaScript : Important aspects

3 *Automatic type coercion*

Operands are converted to the type supported by the operator. Some operators support multiple types

- `true + false` // 1
- `1 + 1 + “2”` // “22”
- `12 / "6”` // 2
- `1 == “1”` // true
- `!!"false" == !!"true”` // true

Characteristics of JavaScript : Important aspects

4 ***« this » keyword***

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

  setInterval(function growUp() {
    that.age++;
  }, 1000);
}
```

Characteristics of JavaScript : Important aspects of ES6

5 *Arrow Function Expression*

- Shorthand notation for regular function expression
- Doesn't have its own bindings to **this**, **arguments**, **super** keywords

```
const sumTwo = (n1, n2) => n1 + n2;
```

```
function sumTwo(n1, n2) {
  return n1 + n2;
}
```

Characteristics of JavaScript : Important aspects of ES6

6 *Classes*

```
class Human {
  constructor(age) {
    this.age = age;
  }

  age() {
    return this.age;
  }
}
```

```
class Person extends Human {
  constructor(name, age) {
    super(age);
    this.name = name;
  }

  get name() {
    return this.name;
  }
}

let someone = new Person('Luke', 34);
someone.age() // "34"
```

Variables (var vs let vs const) – Scope & Hoisting

Var

- Function scoped
- *Undefined* when accessing a variable before it's declared

Let

- Block scoped
- *ReferenceError* when accessing a variable before it's declared

Const

- Block scoped
- *ReferenceError* when accessing a variable before it's declared
- Cannot be re-assigned

Hoisting

- Variable, function and class **declarations** are “moved to the top” of their scope.
- **var** – initialized to *undefined,* while **let** & **const** are *uninitialized.*

```
console.log("Hello", name) // "Hello undefined"
var name = "Michelle";
```

Variable declaration hoisted

```
var name; // var initializes to 'undefined'
console.log("Hello", name); // "Hello undefined"
name = "Michelle";
```

```
console.log("Hello", name) // ReferenceError: name is not defined
let name = "Michelle"; // same thing with const
```

Variable declaration hoisted

```
let name; // let is unitialized
console.log("Hello", name); // ReferenceError
```

Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**)

```
function createCounter() {
  let counter = 0;
  return function() {
    counter = counter + 1;
    return counter;
  }
}

const increment = createCounter();

increment(); // "1"
increment(); // "2"
```

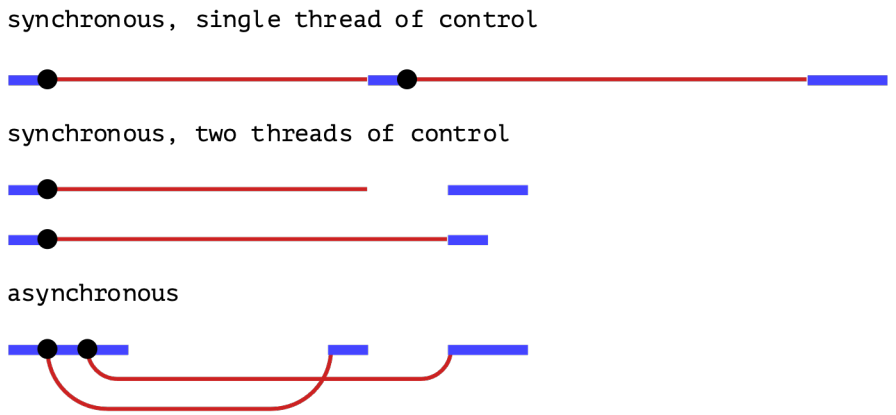
```
function createMultiplier(n1) {
  return function multiply(n2) {
    return n1 * n2;
  }
}

const multiplyByTwo = createMultiplier(2);

multiplyByTwo(3); // "6"
multiplyByTwo(4); // "8"
```

Asynchronous JavaScript

- At its core, JavaScript is a **synchronous**, **blocking**, **single-threaded** language
- That means that only one operation can be in progress at a time
- However, there are ways of working with JavaScript that make it *behave* asynchronously
- 3 common approaches to **asynchronous** JavaScript:
  - Callbacks
  - Promises
  - Async / Await



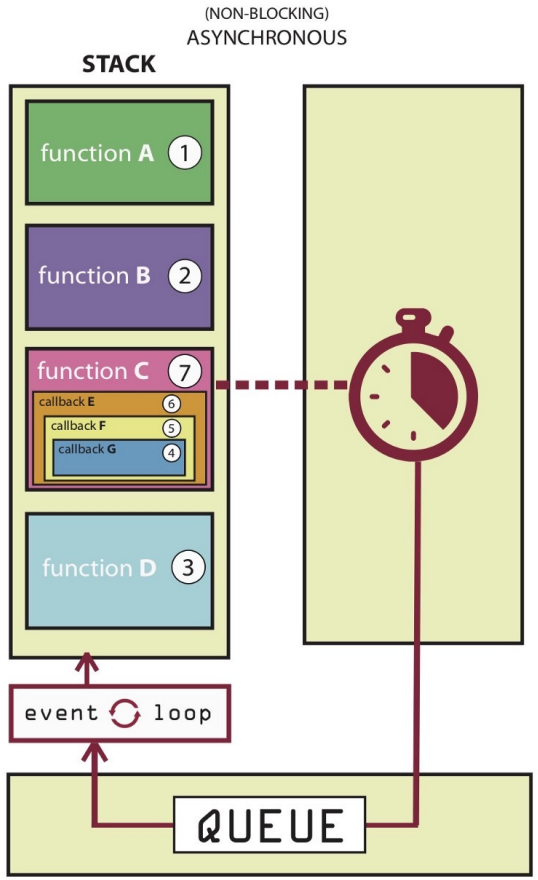
Asynchronous JavaScript (Callbacks)

A **callback** is a function that gets passed to another function as a parameter, which can then be executed as necessary

```
function callApi(url, callback) {
  // some code to fetch data from {url}
  callback(data);
}

function onSuccess(data) {
  // do something with data
}

callApi('http://some-api/v1', onSuccess);
```

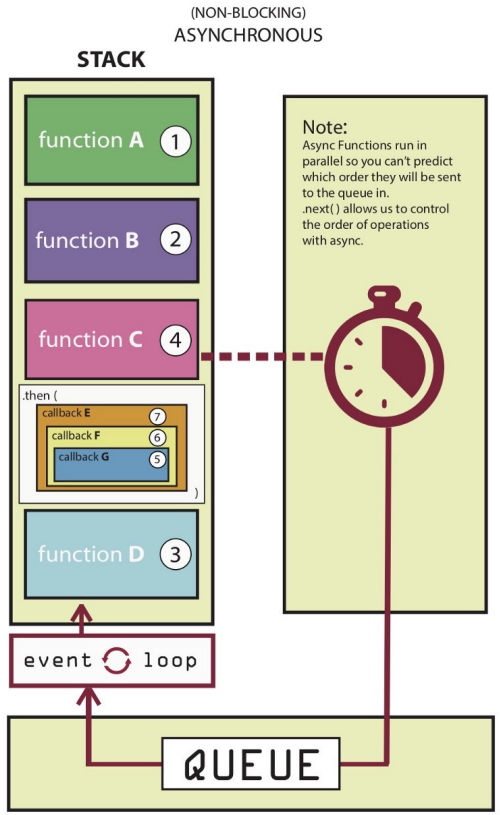


Asynchronous JavaScript (Promises)

A **Promise** is a special object that allows you to invoke asynchronous operations (such as HTTP call) with a ”promise” of their eventual resolution (or rejection, in case of an error)

```
const promiseObj = fetch('http://some-api/v1’);
// "fetch" returns a Promise object

promiseObj
  .then(response => {
    // do something with "response"
  })
  .catch(error => {
    // do something with "error"
  });
```

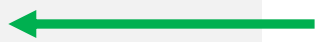


Asynchronous JavaScript (Async / Await)

- The **async / await** notation is another way of working with Promises in ES6
- Lets you write asynchronous code that looks and feels synchronous
- Cleans up your syntax and makes your code more human-readable

```
async function callApi(url) {
  try {
    const response = await fetch(url);
    // do something with response
  } catch (error) {
    // do something with "error"
  }
}

callApi('http://some-api/v1');
```



NOTE that this line won't be executed until **fetch** returns a result

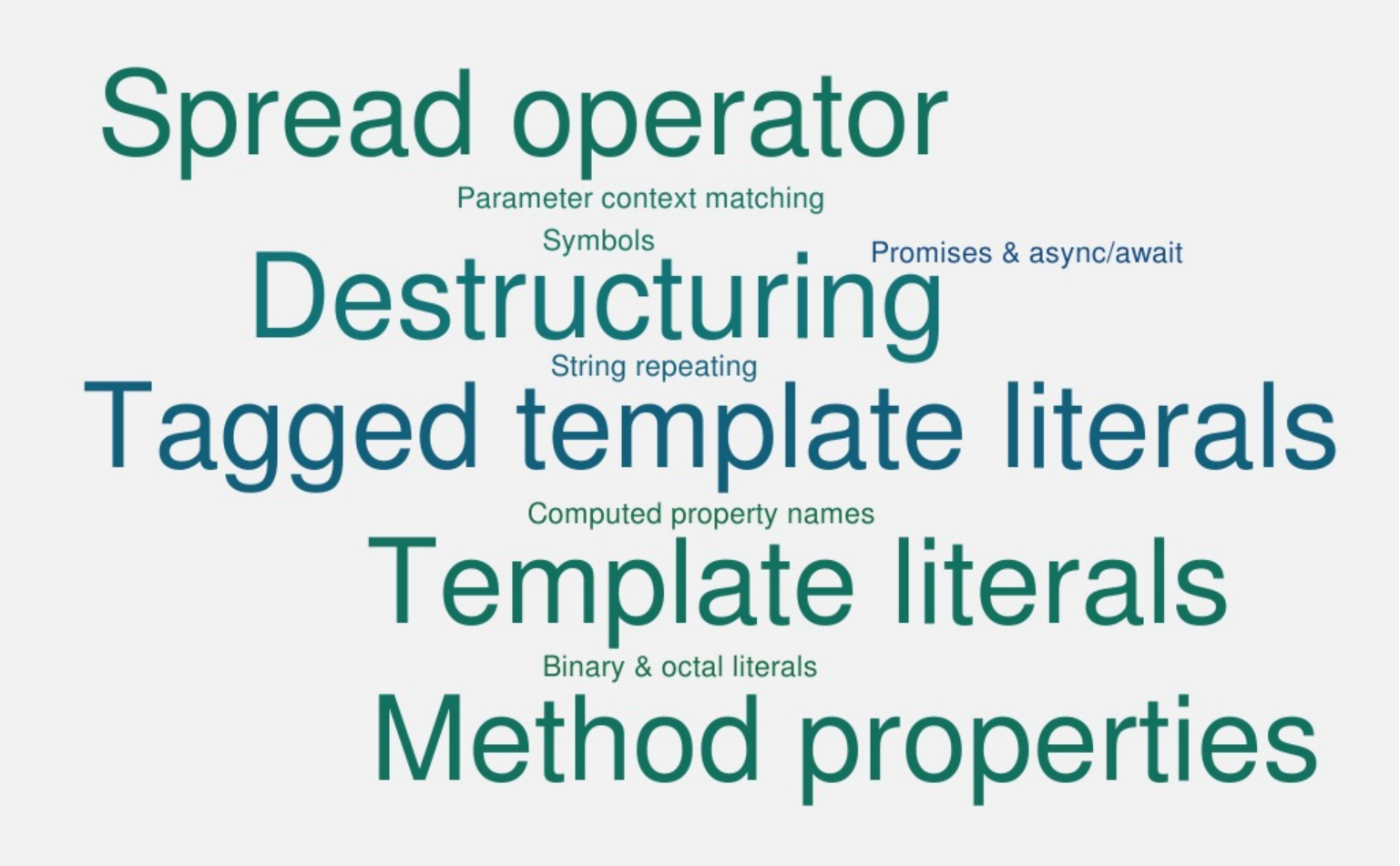
Characteristics of JavaScript

Common concepts

- Single Threaded
- Scope & Hoisting
- Strict Mode
- Modules
- IIFE
- Currying
- Callback

Characteristics of JavaScript

ES6 features



More at: **http://es6-features.org/**

Exercise

Part #1: Warming up (by creating a “Hello World!” NodeJS app)

1. Create a directory for the code-along exericse:
   - ***mkdir <student-ID>-modernjs***
2. Change to that directory:
   - ***cd <student-ID>-modernjs***
3. Initialize NPM package by running the following command:
   - ***npm init***
   - Follow the prompts on the screen (you can leave most values as default), and in the end you enter “***yes***”
4. After that, you should have a new file created called package.json - this is where your project dependencies, scripts, and other info will be specified.
5. Next, let's install some 3rd party dependencies that we'll be using in this exercise:
   - ***npm install esm node-fetch***
   - esm: ECMAScript module loader - this is needed to support ES module syntax in Node
   - node-fetch: Fetch API - this enables the window.fetch() API in Node, used to invoke HTTP requests
6. In the root of your project (<student-ID>-modernjs), create a directory for the source code called src
7. Inside this src directory, create a file for the app entry point named index.js

Exercise

Part #1: Still warming up...

8. Let's add some JavaScript to this file! Type in:
    - ***console.log('Hello World!');***
9. Next, add a start NPM script for the app - this will be used to run our code and see output in console
10. In package.json, add the following to the scripts section:
    - ***"start": "node src/index.js"***
    - This will run the file we created src/index.js using **Node.js** runtime
11. Start the app using the above start script:
    - ***npm start***
    - **NOTE**: start is a reserved script name, so you can execute it without using the run keyword. For other (custom) scripts, you would run them using npm run <script-name>
12. You should see the **Hello World!** message in the console
13. Stop the app using ***Control+C*** (Windows) or ***Cmd+C*** (Mac)
14. Follow the code-along to write the functionality in src/index.js

Exercise

Part #2: Code-Along w/ the coach

- Variable declarations and Scope
- Hoisting
- Objects (initializing, spread operator, destructuring)
- Arrays (initializing, spread operator, rest parameter)
- Functions (Regular vs arrow notation)
- Array methods
  - map, filter, push, find, reduce
- Classes
- Promises
- Async/await
- Import/export

References

Variable declarations

- var - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var
- let - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
- const - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

Functions

- Regular function - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function
- Arrow functions - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
- Function prototype - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/prototype

Objects and Classes

- Classes - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class
- Spread Operator - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

References

Array helper methods

- Map - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- Filter - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- Reduce - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

Modules

- Import - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
- Export - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

Asynchronous Operations

- Promise - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Async - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
- Await - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

References

Babel

- Babel is a transpiler for JavaScript best known for its ability to turn ES6 ("Modern" JavaScript, aka **ES2015** / **ECMAScript2015**) into code that runs in your browser (or on your server) today. Which runs in just about any JavaScript environment.
- Babel can handle all of the new syntax that ES6 brings, along with builtin support for React's JSX extensions and Flow type annotations.
- Of all the ES6 transpilers, Babel has the greatest level of compatibility with the ES6 spec, even exceeding the much longer established Traceur by Google.
- Babel lets you use virtually all of the new features that ES6 brings today, without sacrificing backwards compatibility for older browsers. It also has first class support for dozens of different build & test systems which makes integration with your current toolchain very simple. Babel doesn't only track ES6, it also tracks the next versions of JavaScript (ES7 and beyond). Amongst other things this brings support for async/await, two keywords that are already radically changing the way people write asynchronous JavaScript. They allow developers to write greatly simplified code which is far easier to debug and reason about than when using callbacks, or dealing directly with promises.
- As Babel aligns closely with TC39 (the technical committee that leads the design and development of JavaScript), it is in the unique position of being able to provide real-world-usable implementations of new ECMAScript features before they are standardized.
- Go to https://babeljs.io/ for more information.
- **NOTE:** We are not setting up Babel during the code-along, however it is **highly recommended** to read the above material to familiarize yourself. As a bonus, try adding it to the exercise project yourself! Steps can be found below.

References

Adding Babel config to your project (optional)

- Install Babel as a dependency:
- npm install --save-dev @babel/core @babel/cli @babel/preset-env @babel/node
  - @babel/core: Babel compiler core
  - @babel/cli: Babel command line ("babel" command)
  - @babel/preset-env: Smart preset for Babel that manages JavaScripts syntax transforms
  - @babel/node: CLI ("babel-node" command) to run Node while transpiling on-the-fly (NOT for production use)
- Create a file named .babelrc (Windows magic: name the file .babelrc. to make it work! - or use command line) with the following contents:
- { "presets": ["@babel/preset-env"] } Note above that we had to instruct Babel on which preset to use (and installed that preset as an NPM package)