Intermediate JavaScript

> While we wait...

```
# Download the labs from:
# https://github.com/rm-training/webdev-2021

# Make sure you have node 13+:
node --version

# Install the dependencies:
npm install

# Run a test
npm run test:watch public/solutions/promises
```

Introductions

- What is your experience with JavaScript?
- Specific goals for this class?

Our Goals

> To get *really good* at the fundamentals

- Hoisting
- Coercion
- Scope
- Context
- The prototype

> And to dig into the world of JS in the web

Outline (3 Day)

Day 1

- Revisit the fundamentals
- Higher Order Functions
- Modules
- Objects (OO / Prototype / Classes)

Day 2

- The DOM
- Handling events
- Promises
- Fetch (Ajax)
- Web Sockets

Day 3

- Web Components
- Testing JS (Jest)
- Intro to Tooling

Approaching JavaScript

> Reminders?

- Single-threaded
- No strict types
- Interpreted
- Prototype-based inheritance (vs. class-based)
- No built-in file access; limited I/O; safe sandbox in the web
- Weird but fun 🤪

Versions...

See what's new...

- ES3 - 1999
- ES5 - 2009
- ES6 (ES2015) `<-- we're around here`
- ES2016 - 7th Edition
- ES2017 - 8th Edition
- ES2018 - 9th Edition `<-- rest & spread`
- ES2019 - 10th Edition
- ES2020 - 11th Edition `<-- null coalescing`
- ES.Next

How do you know which version to use?

> Write modern, transpile down to your oldest supportable platform

- Environment
- Device/Platform (ex: Browser)
- Target Audience
- caniuse.com

Resources

- You Don't Know JS
- https://javascript.info/
- Mozilla

Next Up

The Fundamentals

```
8 + 2;     // ?
8 + "2";   // ?
8 - "2";   // ?

8 * null; // ?

[] + [];   // ?
[] - [];   // ?
```

Coercion

- JavaScript is *loosely* typed
- Converts values *on the fly* based on operators at play

```
8 + 2; // 10
8 + "2"; // "82"
8 - "2"; // 6 😲
8 * null; // 0 😲

null > 0; // false
null === 0; // false
null >= 0; // true 😱

[] + []; // "" 🤢
[] - []; // 0  🤢
```

```javascript
+"5"; // 5

let val = "A Bit Too Clever?";
!!val; // true

val = "";
!!val; // false
```

Equality Operators

> A common cause of bugs & confusion

```
// loose - allow coercion
"1" == 1; // true
[3] == "3"; // true
[3] == 3; // true

// strict - prevent coercion
"1" === 1; // false
[3] === "3"; // false

// most strict (ES6+) - fixes NaN comparison
Object.is(1, "1"); // false
```

Truthy & Falsy

> If you use a value as a boolean, it will be *coerced* to a boolean.

Things that are `false`:

```
false;
null;
undefined;
(""); // The empty string
0;
NaN;
```

Everything else is `true`, including:

```
"0";          // String
"false";      // String
[];           // Empty array
{};           // Empty object
Infinity;     // Yep, it's true
```

Optional Chaining

```js
const user = {};

// commonly see this...
if (user && user.address && user.address.id) {
  console.log('I have an address', user.address);
}

// new
if (user?.address?.id) {
  console.log('I have an address', user.address);
}
```

Question: Is there anything troublesome here?

```
const elements = document.querySelectorAll(".titles");

if (elements) {
  console.log("Processing all elements!");
  // ...
} else {
  console.log("No elements found");
}
```

Next Up

Scope

There are three ways we can declare a variable within a `scope`:

- `let`
- `const`
- `var`

Question: What scope are we in?

```
var x;

const user = {};

let dog = 'Fido';
```

Global Scope

> Variables declared in global scope are available to all inner scopes.

Question: What scope `x` are we accessing?

```
var x = 10;

function hello() {
  x = 1;
}

hello();
```

Inner scopes can access outer scopes

Question: What is the scope of `x` and `y`?

```
function add10(x) {
  var y = 10;
  return x + y;
}

hello(12);
```

Question: Will this work?

```
function add10(x) {
  var y = 10;
  return x + y;
}

hello(12);

console.log(x, y); // ?
```

Outer scopes can't access inner scopes

Question: What is the scope of `cat` and `dog`?

```
function catDog() {
  var cat = 12;

  if (cat > 0) {
    var dog = 10;

    if (dog < 100) {
      dog = dog + cat;
    }
  }
}
```

> `var` is scoped to the `function`

```
function catDog() {
  var cat = 12;
  var dog; // <-- hoisting

  if (cat > 0) {
    dog = 10;

    if (dog < 100) {
      dog = dog + cat;
    }
  }
}
```

> And `dog` was *hoisted*

Question: Now what is the scope of `cat` and `dog`? Anything hoisted?

```
function catDog() {
  let cat = 12;

  if (cat > 0) {
    let dog = 10;

    if (dog < 100) {
      dog = dog + cat;
    }
  }
}
```

`let` and `const` are scoped to the block

they are not hoisted

Question: Which `x` is being accessed?

```
let x = "";

function hello() {
  x += "hello";
}

function world() {
  x += " world";
}

hello();
world();
```

Question: Will this work OK?

```
function hello() {
  let x += "hello";
}

function world() {
  let y = "world";
}

hello();
world();

let phrase = `${x} ${y}`;
```

Question: Can we access adjacent scopes?

```
function hello() {
  let x = "hello";
}

function world() {
  hello();

  // is this OK?
  let y = `${x} world`;
}

world();
```

Outer scopes can't access *inner* or *adjacent* scopes

Question: Which `x` is being accessed from the `hello()` function?

```
let x = 10;

function hello() {
  let x = 1;
}

hello();
```

An inner scoped variable with the same name as an outer is said to be *shadowing* the outer.

Question: What if we use `var` in a block?

```
var x = 10;

if (x < 100) {
  // and here?
  var x = 1;
}
```

> Will be interpreted as...

```
var x = 10;
var x; // <-- hoisted the var declaration

if (x < 100) {
  // and here?
  x = 1;
}
```

Question: Will any lines produce an error here?

```
const x = 5;
const y;
const apple = { color: "red" };

x = 12;
apple.color = "blue";
```

`const` creates a constant variable reference, not a constant *value*

`const` objects are still mutable

Question: What are the scopes of `x`, `y`, `z` and `result` here? Will it run?

```
var x = 10;
var y = 11;

function hello(someValue) {
  var y = 20;

  if (x < y) {
    var z = 30;
  }

  return function () {
    var result = {
      value: x + y + z,
    };
    return result;
  };
}

hello(200)(); // returns a fn()
```

Question: What happens if we change these all to `let`, will it run?

```
let x = 10;
let y = 11;

function hello(someValue) {
  let y = 20;

  if (x < y) {
    let z = 30;
  }

  return function () {
    let result = {
      value: x + y + z,
    };
    return result;
  };
}

hello(200)();
```

Question: What will be logged to the console?

```
function init() {
  x = 10;

  var x;

  console.log(x); // ?
}

init();
```

Question: And this time?

```
function init() {
  console.log(x); // ?

  var x = 10;
}
init();
```

Hoisting

- `var` declarations are will be *hoisted* to the top of *function blocks*
- `let` and `const` are *not* hoisted

Function Hoisting

- `function` *statements* are hoisted, too

```
statement(); // valid
expression(); // Error!

function statement() {}
var expression = function () {};
```

Scope Summary

> Determines what variables you can see and access

- Lexical (*as opposed to Dynamic*)
- Global, Function (`var`) and Block (`let`, `const`) scope
- `var` and function statments are *hoisted*
- `const` is an immutable *reference*

Before we get to our first lab...

- Set up for labs?
- Tooling (NPM + Node)
- Setting up ES Lint / Prettier
- Transpiling / Building?

Exercise: Scope

> Messing about in the global scope

1. Open `public/labs/scope/index.js`
2. Do the exercise
3. To test and debug, open `http://localhost:3000/labs/scope/`

Post-exercise discussion

- Blocking the thread?
- Did you use any `const` or `var`?
- What are some pitfalls with this code?
- Would it be easy to test?
- How can we improve it...?

Next Up

Functions

First-class Objects

Three ways to define a function

- Statement
- Expression
- Arrow

The Statement

Reminder: It is hoisted within its scope

```
function add(a, b) {
  return a + b;
}
```

The Expression

Anonymous Functions

```
const add = function (a, b) {
  return a + b;
};
```

```
let recurser = function recursive() {
  recursive();
};
```

Arrow Functions

Mostly relevant when dealing with `context`.

```
const add = (a, b) => {
  return a + b;
};

const terseAdd = (a, b) => a + b;
```

Function Defaults

```
function oldWay(x, y) {
  x = typeof x === "undefined" ? 1 : x;
  y = typeof y === "undefined" ? 10 : y;

  // this works, too! (typeof is more resilient)
  x = x === undefined ? 1 : x;
  y = y === undefined ? 10 : y;

  return x + y;
}
```

Relatively new to JS, you can now define default values.

```
function newWay(x = 1, y = 10) {
  return x + y;
}
```

```
const pleaseInit = (previousVal) => previousVal + 3;

const doodle = function (x, y = 12, z = pleaseInit(y)) {
  return x + y + z;
};

doodle(5); // 5 + 12 + 15 = 32
```

Function Arguments / Arity

```
const add = function (a, b) {
  return a + b;
};

add.length; // 2 - the arity

// these all execute the one `add` function
add(3, 4); // 7
add(1, 2, 3, 4, 5); // 3
```

Consider this...

```
const animals = ["sloth", "peacock", "lion"];

function phraseMe(arr = []) {
  arr.push("humans");
  console.log(`${arr.join(",")} are animals`);
}

console.log(animals.length); // ?
```


Sometimes this is great, sometimes not so much. Definitely not very *functional*.

```
console.log(animals); // ["sloth", "peacock", "lion", "humans"];
```

All Objects are passed by reference

- Objects
- Arrays
- Functions
- Class instances

Question: What will these return?

```
const add = function (a, b) {
  return a + b;
};

add(); // ?
add(1); // ?
add(1,2,3,4,5); // ?
```

Dynamic arguments

`arguments` is a special property available in all functions, but it isn't a real `Array`

```
const addHandlerToElements = function () {
  // converting a array-like thing to an array...
  let elements = Array.prototype.slice(arguments);

  // or use the new Array.from() method to convert...
  elements = Array.from(arguments);

  // so we can do array stuff
  elements.forEach((function(el) => {
    el.addEventListener('click', (e) => {});
  }));
};
```

Use the `...` (rest) operator instead

```
const addHandlerToElements = function (...elements) {
  elements.forEach((function(el) => {
    el.addEventListener('click', (e) => {});
  }));
};
```

Spread in a function call

```
let max = function (x, y) {
  return x > y ? x : y;
};

let ns = [42, 99];

max(...ns); // 99
```

Spread to merge

```
const arr1 = [1,2,3];
const arr2 = [4,5,6];
const merged = [...arr1, ...arr2]; // [1,2,3,4,5,6]
```

```
const obj1 = {id: 1, name: 'mephistopheles'};
const obj2 = {id: 5, species: 'barney'};
const merged = {...obj1, ...obj2}; // {id: 5, name: 'mephistopheles', species: 'barney'}

// Object.assign() is another good merge option
const altMerged = Object.assign({}, obj1, obj2);
```

Destructuring Arrays

```
const arr1 = [1,2,3];

const [x, ,z] = arr1; // x = 1, z = 3

function sum([x,y,z]) {
  return x + y + z;
}
sum(arr1);

// and as we saw before:
// passes 1, 2, 3 to a function argument position (x, y, z)
someOtherFunction(...arr1);
```

Destructuring Objects

```
const user = {
  id: 5,
  name: 'Jim',
};

const {name} = user;

console.log(name); // "Jim"
```

Destructuring objects to function parameters

> Kinda like named function arguments

```javascript
const user = {
  id: 5,
  name: "Tigger",
  isFriendly: true,
};

function updateUser({ id, isFriendly = false }) {
  console.log(`I am going to update user ${id}`);
}

updateUser(user);
```

Aliasing

```
const user = {
  id: 5,
  dogs: [{name: 'Sparky'}, {name: 'Fido'}],
};

function fetchDetails({ id: user_id, dogs: [,secondDog, ...otherDogs] }) {
  return {
    user_id, // 5
    secondDog, // {name: 'Fido'}
    otherDogs, // []
  }
}
```

Exercise: Structure / Destructure

1. Open `public/labs/destructure/index.js`
2. Do the exercise
3. To test and debug, run

   ```
   npm run test:watch public/labs/destructure
   ```

Next Up

Callbacks

Passing a function as an argument to be invoked elsewhere/later

```
function add(a, b, afterSumCb) {
  const sum = a + b;

  afterSumCb(sum);

  return sum;
}

add(1, 3, function (sum) {
  console.log(`I got ${sum}`);
});
```

This is an example of a *higher-order function*.

Higher-order Functions

Functions are a values that we can pass around.

Functions that take other functions, or return new functions, are "higher order" functions.

```
let a = [1, 2, 3];
a.forEach(function (val, index, array) {
  // Do something...
});
```

Functional JS 101

> Break down our code into reusable, easy to test functional components.

To get started, just think: write functions, not for loops.

Go from this:

```
const movies = [
  {"name": "Alien", "rating": 5},
  {"name": "Alien 3", "rating": 4},
  {"name": "Baby Geniuses", "rating": 0},
  {"name": "Beverly Hillbillies", "rating": 2}
];

const topRatedMovies = [];

for (let i = 0; i < movies.length; i++) {
  if (movies[i].rating >= 4) {
    topRated.push(movies[i]);
  }
}
```

To this:

```
const topRatedMovies = movies.filter((movie) => {
  return movie.rating >= 4;
});
```

And then...

```
const isTopRatedMovie = movie => movie.rating >= 4;

const topRatedMovies = movies.filter(isTopRatedMovie);
```

We're writing flexible, reusable, testable, composable code that is easy to reason about.

Higher Order Array Functions

- `every`
- `some`
- `filter`
- `map`
- `reduce`

And more

Array Testing

Test if a function returns `true` on all elements:

```
let a = [1, 2, 3];
a.every(function (val) {
  return val > 0;
});
```

Test if a function returns `true` at least once:

```
a.some(function (val) {
  return val > 2;
});
```

Filtering an array

```
const numbers = [10, 7, 23, 42, 95];

const isEven = function (n) {
  return n % 2 === 0;
};

const even = numbers.filter(isEven);

even; // [10, 42]
even.length; // 2
numbers.length; // 5
```

Mapping over an array

```
const strings = [
  "Mon, 14 Aug 2006 02:34:56 GMT",
  "Thu, 05 Jul 2018 22:09:06 GMT",
];

const dates = strings.map(function (s) {
  return new Date(s);
});

dates; // [Date, Date]
```

Reducing an array

```
const a = [1, 2, 3];

// Sum numbers in `a'.
const sum = a.reduce(function (acc, elm) {
  return acc + elm;
}, 0); // initial value of accumulator

sum; // 6
```

Exercise: Higher Order Array Functions

1. Open the following file:

   `public/labs/array/index.js`

2. Complete the exercise and pass the tests

3. Run the tests:

   - `npm run test:watch public/labs/array`

Hint: Use https://developer.mozilla.org/ for documentation.

Next Up

Functions as Timers

Built-in functions that can establish delays:

```
let timer = setTimeout(() => {
    console.log('I was delayed');
}), 500); // delay in ms

// cancel a timer
clearTimeout(timer);
```

...and intervals:

```
let interval = setInterval(() => console.log('In an interval')), 1000);

// cancel an interval
clearInterval(interval);
```

Question: What will this output?

```
for (var i = 0; i < 3; i++) {
  setTimeout(function () {
    console.log(i);
  }, 1000 * i);
}

console.log("Howdy!");
```

Traditional solution

```
// or... a new function to retain scope access
for (var i = 0; i < 3; i++) {
  (function (j) {
    setTimeout(function () {
      console.log(j);
    }, 1000 * j);
  })(i);
}

console.log("Howdy!");
```

Modern solution

```
// use let
for (let i = 0; i < 3; i++) {
  setTimeout(function () {
    console.log(i);
  }, 1000 * i);
}

console.log("Howdy!");
```

Closures

- Extremely common in JavaScript
- Provides access to an outer function's scope from an inner function
- Created any time you write a function, at runtime

```
let makeCounter = function (startingValue) {
  let n = startingValue;

  return function () {
    return (n += 1);
  };
};

let counter = makeCounter(0); // <-- closure is created when invoked
counter(); // 1
counter(); // 2
```

Why closures?

- Maintain access to a scope (regardless of where they are invoked)
- Simulate privacy
- Maintain state
- "Old World" modules

```javascript
let Foo = function () {
  let privateVar = 42;

  return {
    getPrivateVar: function () {
      return privateVar;
    },
    setPrivateVar: function (n) {
      if (n) {
        privateVar = n;
      }
    },
  };
};

let x = Foo();

x.privateVar; // <-- not available as a property! undefined
x.getPrivateVar(); // 42
```

Exercise: Closures

1. Open `public/labs/closure/index.js`
2. Do the exercise to pass all tests
3. Run the tests:
    - `npm run test:watch public/labs/closure`

Post Exercise Discussion

1. What if I had used `this` and set the temperatures directly on the `object` I exposed?
2. Can I make this a `class` instead?
3. We're actually using a modern ES Module, could we improve the API of our Library?

Next Up

Modules in JS

Modules

Break down large programs into small, manageable and organized components

Encapsulation and reusability.

Traditionally (when all we had was the global scope)...

```
function tempTracker() {
  const temps = [];

  return {
    // setTemp: function (temp) {}
    setTemp(temp) {
      if (temp >= 0 && temp <= 100) {
        temps.push(temp);
      }
    },
    // getTemp: function () {}
    getTemp() {
      return temps[temps.length - 1];
    },
  };
}
const MyModule = tempTracker();
```

One step further...

```
const MyModule = (function() {
  const temps = [];

  return {
    // setTemp: function (temp) {}
    setTemp(temp) {
      if (temp >= 0 && temp <= 100) {
        temps.push(temp);
      }
    },
    // getTemp: function () {}
    getTemp() {
      return temps[temps.length - 1];
    },
  };
}());
```

The IIFE

```
(function () {
  // my modular code...
})();
```

Bag of methods

Also OK, but lacks privacy

```
const MyModule = {
  temps: [],
  getTemp: function() {},
  setTemp: function() {},
};
```

Modules in JS

- Traditionally we used an IIFE
- Or namespace it with an `{}`
- AMD (RequireJS), UMD, CommonJS (node)
- ES Modules (Modern)

Maps and Sets

```
let map = new Map();

map.set('1', 'str1');   // a string key
map.set(1, 'num1');     // a numeric key
map.set(true, 'bool1'); // a boolean key

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

Read up or Here

Exercise: Hosts

1. Open `public/labs/hosts/index.js`
2. Do the exercise to pass all tests
3. Run the tests:
    - `npm run test:watch public/labs/hosts`

Modern Modules..

Static class / singletons

```
class MyModule {
  #temps = []
  static getTemp() {}
  static setTemp() {}
}
```

Common JS (Node)

`TempModule.js`

```
const getTemp = function() {};
const setTemp = function() {};

module.exports.getTemp = getTemp;
module.exports.setTemp = setTemp;
```

`app.js`

```
const tempModule = require('./TempModule.js');
// or
const {getTemp, setTemp} = require('./TempModule.js');
```

ES Modules

`myLib.js`

```
// ...
export { util, sum, add, thing };
export default () => {};
```

`index.js`

```
// importing by name
import { util } from "./myLib.js";

// importing the default
import emptyFunction from "./myLib.js";
```

`myLib.js`

```
export { util, sum, add, thing };
```

`index.js`

```
import * as SomeName from "./myLib.js";

SomeName.util(); // works!
```

ES Modules

- It's own scope
- `export` and `import`
- "named" exports or a `default`
- Not totally supported (but almost)
- supports circular dependencies

I have a demo

Live Binding

`thingy.js`

```
let x; // I'm shared
export default {
  get() {
    return x;
  },
  set(val) {
    x = val;
  }
}
```

`index.js`

```
import thingy from './thingy.js';

thingy.set(10);
```

`otherMod.js`

```
import thingy from './thingy.js';

thingy.get(); // 10
```

Demo

Exercise: Modernize Hosts

1. Update `public/labs/hosts/index.js`
2. Convert fully to a modern ES Module
    - Tests should all still pass
    - You'll need to make *one* tweak to the tests... in the `import`
3. Run the tests:
    - `npm run test:watch public/labs/hosts`

Next Up

Objects and Context

Question: Identify the *code smell*?

```
const human = {
  name: "Ryan",
  sayHello: function () {
    console.log(`${human.name} says: "Hello"`);
  },
};
```

```javascript
const human = {
  name: "Ryan",
  sayHello: function () {
    console.log(`${this.name} says: "Hello"`);
  },
};

const cat = {
  name: 'Felix',
  sayHello: human.sayHello,
};
```

```javascript
const sayHello = function() {
  console.log(`${this.name} says: "Hello"`);
}

const human = {
  name: "Ryan",
  sayHello,
};

const cat = {
  name: 'Felix',
  sayHello,
};
```

`this` is about making functions flexible enough to operate on many different objects

Context

`this` references the context of a function

- it's dynamic
- changes based on where it is invoked
- really has no place outside of being an object method

```javascript
const animal = { purpose: "Survive" };
const cat = { purpose: "Nap, in a professional capacity" };
const human = { purpose: "Engineer Things" };

function explainPurpose() {
  console.log(`I am here to: ${this.purpose}`);
}

animal.explain = explainPurpose;
cat.explain = explainPurpose;
human.job = explainPurpose;

animal.explain(); // I am here to: Survive
cat.explain(); // I am here to: Nap, in a professional capacity
human.explain(); // I am here to: Engineer Things
```

Question: What will `this` be here?

```
function explainPurpose() {
  console.log(`I am here to: ${this.purpose}`);
}

explainPurpose(); // ?
```

Controlling context

`call`, `apply` let us set context on the fly

```
const cat = { name: "Jim" };

const speak = function (words) {
  console.log(`${this.name} says: ${words}`);
};

speak.call(cat, "Meow");
speak.apply(cat, ["Meow"]);
```

`bind` creates a new function with a hard-bound context

```
const catSpeak = speak.bind(cat);
catSpeak("Purrrrr");

const catSaysMeow = speak.bind(cat, "Meow");
catSaysMeow();
```

Consider:

```
const timerBot = {
  name: "Timer Bot",
  setTimer() {
    setTimeout(function () {
      console.log(`${this.name} says hello`);
    }, 1000);
  },
};
// what will be logged?
```

Using bind, we can...

```
const timerBot = {
  name: "Timer Bot",
  setTimer() {
    setTimeout(
      function () {
        console.log(`${this.name} says hello`);
      }.bind(this),
      1000
    );
  },
};
```

Arrow functions on the other hand don't have a `context` of their own.

```
const speak = () => {
  // lexically bound...
  console.log(`${this.name} says hello`);
};
```

Arrow functions *are* great, because...

```
const timerBot = {
  name: "Timer Bot",
  setTimer() {
    setTimeout(() => {
      // `this` is the outer function's context
      console.log(`${this.name} says hello`);
    }, 1000);
  },
};
```

Careful...

> Arrow functions are bound based on where they are declared

```
const speak = () => {
  console.log(`${this.name} says hello`);
};

const timerBot = {
  name: "Timer Bot",
  setTimer() {
    setTimeout(speak, 1000);
  },
};
```

Next Up

OO in JS

Creating Objects

- The object literal
- `Object.create()`
- Basic Constructors
- Class Keyword

Object Property Shorthand

```
const legs = 0;
const speak = function() {
  console.log(`${this.name} says hi`);
};

const animal = {
  legs,
  speak,
  walk() {
    console.log("I am walking");
  },
};
```

Prototypal Delegation

`Object.create()` will create a new object with a prototypal link to another object.

```
const animal = {
  legs: 0,
  fur: true,
  walk() {
    console.log("I am walking");
  },
};

const dog = Object.create(animal);
dog.legs = 4;

const mechaDog = Object.create(dog);
mechaDog.fur = false;
```

Constructor Functions and the `new` Operator

Constructor functions, which utilize the `new` keyword, can be used to create object instances that are linked to the constructor's `prototype`

```
function Animal(legs = 0, fur = false) {
  this.legs = legs;
  this.fur = fur;
}

Animal.prototype.walk = function () {
  console.log("I am walking");
};

const dog = new Animal(4, true);
```

Prototype Chain

- Simulates multiple inheritance
- Can't have have more than one "parent" object

```
function Dog() {
  Animal.call(this, 4, true);
}

Dog.prototype = Object.create(Animal.prototype);
```

Exercise: Constructor Calculator

> Don't jump ahead to `class` keyword just yet 😃

1. Open the following file: `public/labs/constructors/index.js`
2. Complete the exercise
3. Run the tests:
    - `npm run test:watch public/labs/constructors`

The Class Keyword

Introduced in ES6 as more concise abstraction for creating objects that delegate to one another.

```
class Animal {
  constructor(legs = 0, fur = false) {
    this.legs = legs;
    this.fur = fur;
  }

  walk() {
    console.log("I am walking");
  }
}

const dog = new Animal(4, true);
```

Extending Classes

```
class Dog extends Animal {
  constructor(color) {
    this.color = color;
    super(4, true);
  }
}

const instance = new Dog();
```

Getters & Setters

Traditionally relied on `Object.defineProperty`

This is *not* for privacy.

```
const user = {
  age: 100
}

Object.defineProperty(user, 'age', {
  get() {
    return "None of your business";
  },
  set(value) {
    this._age = value;
  }
});
```

You can define other properties

```
Object.defineProperty(user, 'birthdate', {
  enumerable: true,
  configurable: false,
  writable: false,
  value: '1981-01-01'
});
```

Getters & Setters (Now)

```
class Car {
  constructor() {
    this._speed = 0;
  }
  get speed() {
    return this._speed;
  }
  set speed(x) {
    if (x < 0 || x > 100) {
      throw "I don't think so";
    }
    this._speed = x;
  }
  static clone() {}
}
var toyota = new Car();
toyota.speed = 55; // Calls the `set speed' function.
```

Statics

```
class User {
  static maxUsernameLength = 64;
  static clone(user) {
   // .. implementation details
  }
}

const ryan = new User();
const rhino = User.clone(ryan);
```

Properties / Fields

Public, private and static properties (fields)

Note: These are still experimental.

```
class Rectangle {
  height = 0;
  width;
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Private Fields

```
class User {
  // private field
  #id;

  constructor(id) {
    this.#id = id;
  }

  save() {
    console.log(`Saving User ${this.#id}`);
  }

  // private method
  #subscribe() {}
}

const me = new User(10);
me.#id; // error
```

Question: What will the context be?

```
const user = {
  me: () => console.log(this)
}
user.me(); // ?
```

Question: What is the difference between `sit` and `run`?

```
class Pet {
  sit = () => {
    this.isSitting = true;
  }
  run() {
    this.isRunning = true;
  }
}
const dog = new Pet();
```

Exercise: Class Upgrade

1. Revisit `public/labs/constructors/index.js`
2. Upgrade your constructor function to use a `class` instead
3. Run the same tests:
   - `npm run test:watch public/labs/constructors`

Behavior Sharing Patterns in JS

> You don't have to use a `class` for that.

- Objects as bag of methods
- Augmentation (Decoration)
- Functional Object Instantiation (Factory-like)
- Object Composition

Augmentation

```
function makePlayable(obj) {
  obj.isPlayable = true;
  obj.play = function() {
    console.log(`${this.name} is playing`);
  }
}

const dog = new Dog('Fido');
makePlayable(dog);
```

A bit more functional

```
function makePlayable(obj) {
  const updatedObj = Object.assign({}, obj);

  updatedObj.isPlayable = true;
  updatedObj.play = function() {
    console.log(`${this.name} is playing`);
  }
  return updatedObj;
}

const dog = new Dog('Fido');
const playableDog = makePlayable(dog);
```

Functional Object Instantiation

> Simple, but you *are* creating lots of copies

```
function User(id, name) {
  return {
    id,
    name,
    save() {
      console.log(`Saving User ${this.#id}`);
    }
  }
}

const me = User(10, "Ryan");
```

Object Composition

> Avoids the "God Object" anti-pattern of traditional inheritance hierarchies

```
User
Animal
  -> Dog
  -> Cat
```

I want to share behavior between `Cat` and `Dog`, so I have an `Animal` class. Now I want to share that with `User`, so I make another higher level class?

```
const eater = (state) => {
  return {
    eat(food) {
      console.log(`${state.name} eats ${food}`);
    }
  }
}
const barker = (state) => {
  return {
    bark(food) {
      console.log(`${state.name} barks`);
    }
  }
}
```

```
function Dog (name, energy, breed) {
  let dog = {
    name,
    energy,
    breed,
  }

  return Object.assign(
    dog,
    eater(dog),
    barker(dog),
  )
}
```

> Like using `this`?

```
function Cat (name, energy, declawed) {
  this.name = name
  this.energy = energy
  this.declawed = declawed

  return Object.assign(
    this,
    eater(this),
    sleeper(this),
    player(this),
    meower(this),
  )
}

const charles = new Cat('Charles', 10, false)
```

Next Up

JavaScript and the Browser

- HTML for the content & structure
- CSS for presentation
- JavaScript for behavior & business logic

HTML Refresher

- Hyper Text Markup Language
- Plain text
- Very error tolerant
- Tree of nodes

```html
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1 id="title">Welcome</h1>
    <p>Awesome <span class="loud">Site!</span></p>
  </body>
</html>
```

HTML Elements

```html
<div key="value" key2="value2">Text content of element</div>

<!-- self-closing -->
<input name="username" />
```

The HTML Tree

Let's look at some pages if needed

CSS

- Cascading Style Sheet
- Rule-based language for describing presentation
- Separate file or inline
- Can handle quite a lot these days:
  - Animation
  - Grids
  - Spatial positioning
  - Variables

What does CSS look like?

```css
#container {
  margin: 5px;
}

p {
  background-color: white;
  color: blue;
  padding: 5px;
}

.spoiler {
  display: none;
}

p.spoiler {
  display: block;
  font-weight: bold;
}
```

CSS Selectors

- Help to specify elements in our page
- Which is key to page manipulation
- Such as:
    - id
    - class
    - element name
    - parent/child relationship
    - combination of the above

How the browser loads the page

- Top to bottom (HTML, JS)
- Loads resources as it comes across them
- Some resources (ie: scripts) can be blocking

```
<script src="somefilename.js"></script>

<script>
  let x = "Hey, I'm JavaScript!";
  console.log(x);
</script>

<button onclick="console.log(x);"></button>
```

Scripts are blocking by default

```html
<script src="somefilename.js"></script>
<h1>Hello?</h1><!-- you won't see this until that ^ script finishes -->
<script src="somefilename.js"></script>
<h2>World?<h2>
```

When your JS loads is important

- You want to avoid a long wait to see content
- You want JS that is critical to run quickly
- You want JS that needs the DOM should run when the DOM is ready

Where you put your scripts matters

- inline `<script>` or attributes
- in the `<head>`
- at the bottom of the `<body>`

Control when a script is downloaded & executed

- `async`
- `defer`
- `type="module"`

```
<script defer src="jquery.js"></script>

<script>
  window.onload = function() {
    console.log('Everything is done!');
  }
</script>
```

The DOM

> What most people hate(d) about JavaScript

- API for the document
- Represents elements as a tree of nodes
- Live data structure

```
const thingyEl = document.getElementById("thingy");
```

Element Nodes

The HTML:

```html
<p id="thingy" class="hi">My <span>text</span></p>
```

Maps *loosely* to:

```js
let node = {
  tagName: "P",
  childNodes: NodeList,
  className: "hi",
  innerHTML: "My <span>text</span>",
  id: "thingy",
  // ...
};
```

Typically working with the DOM will involve

- Select an element to gain access
- Traverse as needed
- Create/Modify/Add behavior

Selecting

```html
<div id="m-id" class="fancy"></div>
<div class="boring"></div>
```

```javascript
let el = document.getElementById("my-id");

// first matching element
el = document.querySelector("#my-id");
el = document.querySelector("div.fancy");

// all matching elements
el.querySelectorAll("div");
```

There is also…

- `getElementsByTagName`
- `getElementsByClassName`

Traversing

Moving between nodes via their relationships

```html
<div class="the-parent">
  <div class="the-child">
    <div>TBD</div>
  </div>
</div>
```

```javascript
let el = document.querySelector(".the-child");

el.children[0].innerHTML = "<h1>Hi!</h1>";
el.parentElement; // <div class="the-parent">...</div>

// or select within...
el.querySelector('h1');
```

Traversal Properties

- `parentElement`
- `children`
- `firstElementChild`
- `lastElementChild`
- `previousElementSibling`
- `nextElementSibling`

There are also things like `nextSibling` and `childNodes`; these are older accessors and may not always give you an `Element` object back.

Node Types

`element.nodeType`

- `1` : Element
- `3` : Text Node
- `8` : Comment Node
- `9` : Document Node

Creating & Appending New Elements

- `createElement`
- `createTextNode`

```
const newEl = document.createElement("h1");
const text = document.createTextNode("Hello");
```

Insertion

Then you'll put it into the DOM tree:

- `el.appendChild(newEl)`
- `el.insertBefore(newChild, existingChild)`
- `el.replaceChild(newEl, existingEl)`
- `el.removeChild(existingEl)`

```
const newEl = document.createElement("h1");
const text = document.createTextNode("Hello");

newEl.appendChild(text);

document.getElementById("some-root").appendChild(newEl);
```

Modifying Elements

You can insert HTML strings, which the browser will parse.

```
el.innerHTML = "<h1>Hello World</h1>";

// can do the same with text nodes
el.textContent = "Hello";
```

Attributes

```html
<div class="user-info" data-user-id="5"></div>
```

```js
el.getAttribute(name);
el.setAttribute(name, value);
el.hasAttribute(name);
el.removeAttribute(name);
```

DataSet API

> Manage some state/data along with an element

```html
<div class="user-info" data-user-id="5"></div>
```

```js
el.dataset.userId;
```

ClassList API

Vanilla JS + the DOM is converging on common patterns.

```
el.classList.add(name);
el.classList.remove(name);
el.classList.toggle(name);
el.classList.contains(name);
```

Exercise: DOM Manipulation

1. Open the following files in your text editor:

```
public/labs/flags/index.js
public/labs/flags/index.html (read only!)
```

2. Start your server with `npm start`
3. Visit http://localhost:3000/labs/flags
4. Complete the exercise while testing in-browser
5. Run the tests:
    - `npm run test:watch public/labs/flags`

Next Up

Events

The Event Loop

> JavaScript is single threaded... so it has a single call stack and can do one thing at a time.

- Events fire
  - click, page ready, focus, submit, scroll, etc...
- Which queues handler functions
- Browser processes funtions in the queue
  - The event loop
  - one function at a time;
  - it is blocking

Demo a Runtime: /demo/runtime/

https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

```
while (queue.waitForMessage()) {
    queue.processNextMessage();
}
```

Don't block the thread

Move non-critical JS out of the main thread on page load...

- `async`
- `defer`
- at the bottom of the page

More on the Handler Queues

- Multiple queues, browser can decide what to do first
- Task Queue (generally all events, but don't count on that)
    - One task per *tick* (loop)
    - `setTimeout(fn)`
- Microtask Queue (promise handlers, for one)
    - Completes all in queue per tick
- Rendering step (`requestAnimationFrame(fn)`)

Go deep: https://vimeo.com/254947206 and This one

Handling Events

1. Select an element
2. Define a handler function
3. Register the handler on the element

```
// define a function
const myFunction = function () {};

// select an element
const el = document.getElementById("container");

// all the handler
el.addEventListener("click", myFunction);
```

Remove handlers

Good practice to remove a handler if you no longer need it (ex: removing the element, hiding it for a long period of time, etc).

```
el.removeEvenListener("click", myFunction);
```

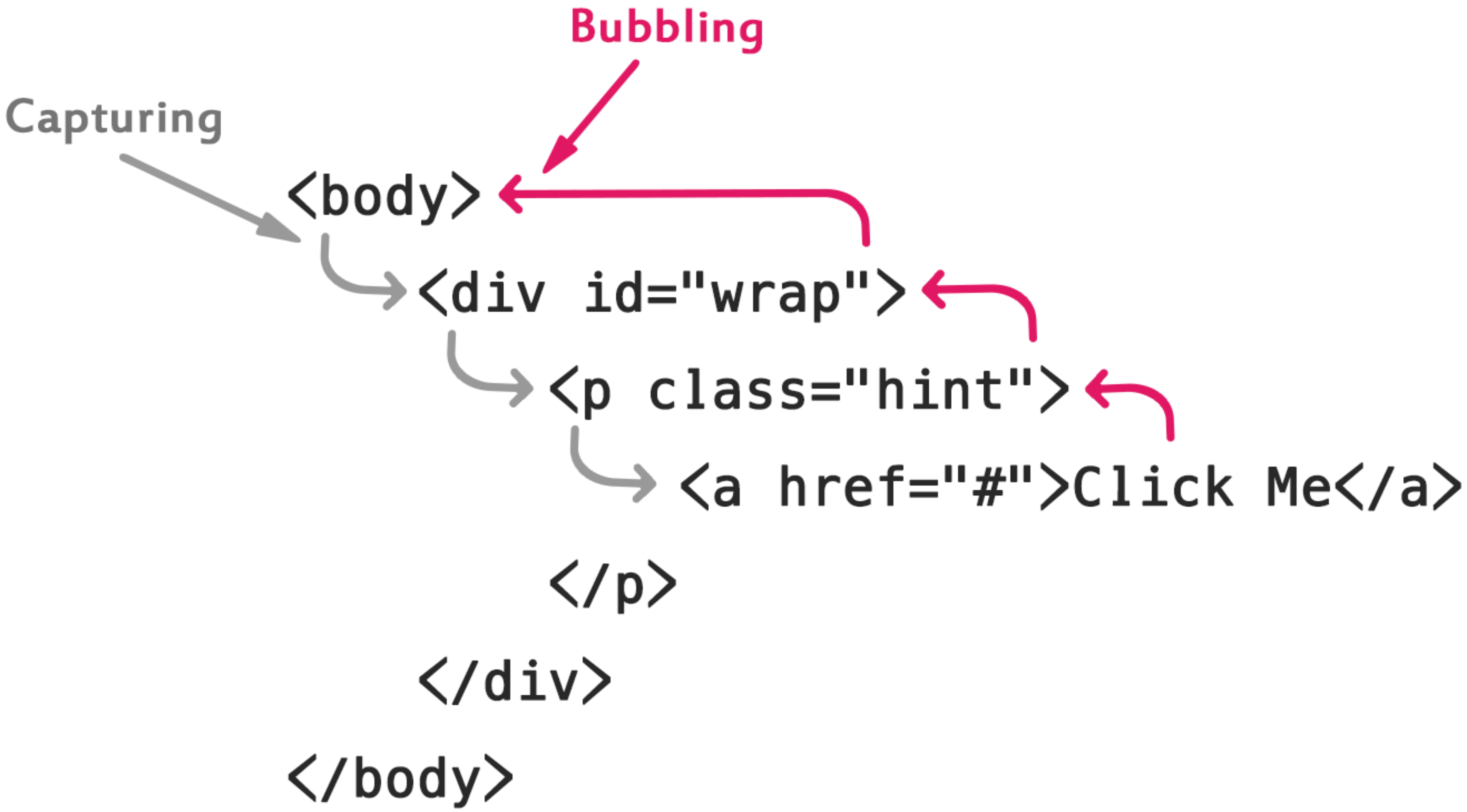
Event Handlers

- Handlers are passed an "event object"
- The context (`this`) is the element where the handler is registered

```
const myFunction = function (eventObject) {
  console.log(this); // where I am registered (traditionally not consistent)

  eventObject.target; // element which triggered the event
  eventObject.currentTarget; // element where handler is registered
};
```

Events Propagate



Controlling the event

```
eventObject.stopPropagation();
eventObject.preventDefault();
eventObject.stopImmediatePropagation();
```

Returning false from a handler will also stop default behavior.

Event Delegation

Using `event.target` and `event.currentTarget` we can have a handler function that manages all the events of a set of child elements.

Example: /demo/events.html

Event Warnings

- Don't block the thread
- Break up long running functions
  - `setTimeout(continueFn, 0);`
- Debounce event handlers

Page Ready Events

We want JS that depends on the DOM to run only once the DOM is fully **parsed**.

- `window` has `load`
- `document` has `DOMContentLoaded`
  - also requires `readyState`

```
const fn = () => {};
window.onload = fn;

window.addEventListener('load', fn);

document.addEventListener('DOMContentLoaded', fn);
```

Question: What is wrong here?

```
const user = {
  id: 1,
  initHandlers() {
    const el = document.querySelector(".user");
    el.addEventListener("click", function () {
      console.log(`User #${this.id} was clicked`);
    });
  },
};

user.initHandlers();
```

Context in Callbacks (3 solutions)

1. use an arrow function
2. Maintain via closure, `const that = this;`
3. Lock in the context, `call()` or `bind()`

```
const user = {
  id: 1,
  initHandlers() {
    const el = document.querySelector(".user");

    el.addEventListener("click", () => {
      console.log(`User #${this.id} was clicked`);
    });
  },
};

user.initHandlers();
```

Context in Callbacks

- When you pass your function to be called elsewhere
  - You can't rely on the **context**!
- Applies to *all callbacks*, not just event handlers

One more gotcha

Question: Spot any problems?

```
if (document.readyState == "complete") {
  initSite();
} else {
  window.onload = initSite();
}
```

Exercise: Simple User Interaction

1. Open the following files in your text editor:

```
public/labs/events/index.js
public/labs/events/index.html (read only!)
```

2. Open [http://localhost:3000/labs/events] in your web browser.
3. Complete the exercise.

Next Up

Promises

> Construct to represent some data you may not yet have (future data)

You can pass this "future data" around!

```
const myData = fetch('/api/artists');

console.log(myData); // Share your Promise
```

What do promises solve for?

Synchronous code is pretty simple to handle...

```
let result1 = getData();
let result2 = getData(result1);
let result3 = getData(result2);
console.log(result3);
```

But what if the functions can't actually return immediately - if they are asynchronous?

A common pattern is to pass in a callback to be invoked upon completion.

```
getData(undefined, function (result1) {
  // when the asynchronous thing is done, run me!
  getData(result1, function (result2) {
    // when the asynchronous thing is done, run me!
    getData(result2, function(result3) {
      // when the asynchronous thing is done, run me!
      console.log(result3);
    });
  });
});
```

Callback Hell / Pyramid of Doom

```
getData(undefined, {
  success: function(result1) {
    getData(result1, {
      success: function (result2) {
        getData(result2, {
          success: function(result3) {
            console.log(result3);
          },
          failure: function(error) {
            console.log('Something went wrong!');
          }
        });
      },
      failure: function(error) {
        console.log('Something went wrong!');
      }
    })
  },
  failure: function(error) {
    console.log('Something went wrong!');
  }
})
```

With Promises...

```
getData()
  .then(function(result1) {
    return getData(result1)
  })
  .then(function(result2) {
    return getData(result2);
  })
  .then(function(result3) {
    console.log(result3);
  });
```

> It doesn't matter if I'm using arrow functions or not...

```
getData()
  .then((result1) => {
    return getData(result1)
  })
  .then((result2) => {
    return getData(result2);
  })
  .then((result3) => {
    console.log(result3);
  });
```

Promises

> Flatten asynchronous code that would otherwise be deeply nested

- States: `Pending`, `Fulfilled`, or `Rejected`
  - Resolved/Settled: No longer `pending`
- Composable

Promise Creator

- Constructs the Promise
- Decides when it is considered "Resolved" and "Rejected"
- Returns some data (when resolved) or an error (when rejected)

```
const delayFor = function (resolveInMs) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("All done!");
    }, resolveInMs);
  });
};
```

Promise Consumer

- `then()`, `catch()`, `finally()`
- Chainable
- Store the promise and pass it around

```
const resolveHandler = (data) => {};
const rejectionHandler = (error) => {};

const prom = delayFor(100);

prom.then(resolveHandler, rejectionHandler);
prom.then(resolvedHandler);

// pass it around
someOtherThingThatWorksWithPromises(prom);
```

Chaining

```
const prom = delayFor(100);

const lastProm = prom.then((data) => {
    console.log(data); // "All done!"
    return delayFor(500);
  }).then((data) => {
    console.log(data); // "All done!"
    // returns are automatically wrapped in a promsie
    return "World";
  }).then((data) => {
    console.log(data); // "World"
  });

// lastProm?
```

Parallel

```
const prom1 = delayFor(100);
const prom2 = delayFor(500);
const prom3 = delayFor(200);
```

Composable

```
// all fulfilled or first rejection
const allDone = Promise.all([prom1, prom2]);

// all resolved in any way
const allSettled = Promise.allSettled([prom1, prom2]);

// first resolved in any way
const firstDone = Promise.race([prom1, prom2]);

// first fulfilled, otherwise rejects
const firstSuccess = Promise.any([prom1, prom2]);
```

Composing / Breaking the Chain

> ex: get user and posts in parallel, then comments after posts, finally do something when all are done...

```
const userPromise = getData("/users/1");
const postsPromise = getData("/users/1/posts");

const commentsPromise = postsPromise.then((posts) => {
  let firstPost = posts[0];
  return getData(`/posts/{$firstPost.id}/comments`);
});

const allTheThings = Promise.all([userPromise, postsPromise, commentsPromise]);

// destructuring an array...
allTheThings.then(([user, posts, comments]) => {
  // now I have everything!
});
```

Handling Errors

- Any error thrown in a promise will be treated as a `rejection`
  - `reject(new Error())` is the same is `throw new Error()`
- Use `.catch` or `.finally`

```
const prom = fetch('/posts');

prom
  .then(undefined, (error) => {
    // something went wrong...
  })
  .then(undefined, (error) => {
    // something went wrong...
  })
  .catch((error) => {
    // catches any previously unhandled errors/rejects
  })
  .finally(() => {
    // cleanup code
  });
```

`catch` can handle all previously unhandled errors/rejects

It also returns a `promise`.

```
const prom = fetch('/posts');

prom
  .then(() => {
    console.log("Step One");
  }))
  .then(() => {
    console.log("Step Two");
  })
  .catch((error) => {
    console.error("Something went wrong but we're OK with that", error);
    return "Not a problem";
  })
  .then((data) => {
    console.log("Step Three");
    console.log(data); // "Not a problem"
  });
```

Promises can (sometimes) swallow errors

Which of these will produce an error in the console?

```
Promise.resolve('promised value').then(function() {
    throw new Error('error');
});

Promise.reject('error value').catch(function() {
    throw new Error('error');
});

new Promise(function(resolve, reject) {
    throw new Error('error');
});
```

Depends in part on the browser implementing the promise spec.

Don't forget to share

Promises are objects that can be passed back, used elsewhere. Ends up being helpful with error handling.

```
const saveUser = (data) => {
  database.save(data)
    .then(() => console.log("Success!"))
    .catch((error) => console.log("Something went wrong!"));
}
```

to...

to...

```
const saveUser = (id, data) => {
  return database.save(data);
}

const clickHandler = (e) => {
  saveUser(e.target.data['user-id'])
    .then(() => UI.success("Saved!"))
    .catch((error) => UI.error("Something went wrong!"));
}
```

Avoid callback hell

```
const request = saveUser(data);

request.then((user) => {
  subscribeUser(user.id).then((subscription) => {
    updateUI(subscription).then(() => {
      console.log('All Done');
    });
  });
});
```

Any value returned from a `then` will be wrapped in a promise.

```
const request = saveUser(data);

request.then((user) => {
  return subscribeUser(user.id)
}).then((subscription) => {
  return updateUI(subscription);
}).then(() => {
  console.log('All Done');
});
```

> **Question**: What will the `value` be?

```
const value = saveUser(data)
  .then((user) => {
    return user.username;
  });
```

Question: Are these promises going to run in parallel or in sequence?

```
// Promise.resolve() immediately resolves
// but it could be just as easily an AJAX request
const prom1 = Promise.resolve(1);
const prom2 = Promise.resolve(2);
const prom3 = Promise.resolve(3);
```

> **Question:** Are these promises going to run in parallel or in sequence?
> Which promise is captured in `prom`?

```
const prom = Promise.resolve(1)
  .then(() => Promise.resolve(2))
  .then(() => Promise.resolve(3));
```

Question: Spot any issues here?

```
const runTracker = {runs: 0};
const delayFor = function (resolveInMs) {
  return new Promise(function (resolve, reject) {
    if (resolveInMs < 0) {
      reject("Bad argument");
    }
    runTracker.runs++;
    setTimeout(function () {
      resolve("All done!");
    }, resolveInMs);
  });
};

delayFor(-50).then((success) => {
  console.log("Good Promise:", success);
}).catch((error) => {
  console.log("Bad Promise:", error);
});
```

Exercise: Promises

1. Open `public/labs/promises/index.js`
2. Implement the functionality to pass tests and get the code running in the browser, too
3. Test with jest:

   ```
   npm run test:watch public/labs/promises
   ```
4. Test in your browser:

   localhost:3000/labs/promises/

Next Up

Loading data with `fetch` / AJAX

Ajax Basics

- Asynchronous JavaScript and XML
  - It is non-blocking!
- API for making HTTP requests
- Originally handled via `XmlHttpRequest` object
- Can be in any format, usually `json`, `html` or `xml`
- `same-origin` policy / CORS

JSON

- String representation of a JavaScript Object
- Not exact -- functions are not represented

```
let object = {
  id: 10,
  name: "Ryan",
  awards: [1, 2, 3], // arrays are OK
  sayName: function () {
    // functions will be ignored
    console.log(this.name);
  },
};
JSON.stringify(object); // "{"id":10,"name":"Ryan","awards":[1,2,3]}"
JSON.parse(string);
```

XHR Object

- The original AJAX
- Inconsistent, lots of boilerplate

```
let req = new XMLHttpRequest();

req.addEventListener("load", function (e) {
  if (req.status == 200) {
    console.log(req.responseText);
  }
});

req.open("GET", "/example/foo.json");
req.send(null); // this is where you could send a form body
```

Fetch API

- New in modern browsers
- Uses **Promises**
- Easily handles file uploads
- No IE (easy to polyfill)

The Fetch Function

Notice how the response provides the json data as another Promise

```
fetch("/api/artists", { credentials: "same-origin" })
  .then(function (response) {
    return response.json(); // <-- take note!
  })
  .then(function (data) {
    updateUI(data);
  })
  .catch(function (error) {
    console.log("Ug, fetch failed", error);
  });
```

Fetch options

```
fetch(url, {
  method: "POST",
  credentials: "same-origin",
  headers: { "Content-Type": "application/json; charset=utf-8" },
  body: JSON.stringify(data),
})
  .then(function (response) {
    if (response.ok) {
      return response.json();
    }
    throw `expected ~ 200 but got ${response.status}`;
  })
  .then(console.log);
```

Fetch Errors

- **fulfills** for any HTTP response, including a 404 or 500
- **rejects** only if request does not resolve (ie: network failure)

Aborting

```
const controller = new AbortController();
const signal = controller.signal;

// invoke `abort()` to manually cancel the request
setTimeout(() => controller.abort(), 5000);

// fetch accepts a `signal` option
fetch(url, { signal }).then(response => {
  return response.json();
}).then(json => {
  console.log(json);
}).catch((error) => {
  if (err.name === 'AbortError') {
    console.log('Fetch aborted');
  } else {
    console.error('Uh oh, an error!', err);
  }
});
```

Exercise: Using the Fetch API

1. Open `public/labs/fetch/index.js`
2. Implement the function

   The API is available at http://localhost:3000/artists
3. Test with jest:

   ```
   npm run test:watch public/labs/fetch
   ```
4. Test in your browser, if you want:

   localhost:3000/labs/fetch/

Next Up

Async & Await

Unwrap "promises" to make asynchronous code read more synchronously

Makes code like this...

```
fetch("/users/1")
  .then((response) => {
    if (response.ok) {
      return response.json();
    }
  })
  .then((user) => {
    return fetch("/users/1/posts");
  })
  .then((response) => {
    if (response.ok) {
      return response.json();
    }
  })
  .then((posts) => {
    return fetch(`/posts/${posts[0].id}`);
  })
  .then((response) => {
    if (response.ok) {
      return response.json();
    }
  });
```

More like this:

```
async function getFirstPost() {
  const userResponse = await fetch("/users/1");
  const user = await userResponse.json();

  const postsResponse = await fetch("/users/1/posts");
  const posts = await postsResponse.json();

  const firstPostResponse = await fetch(`/posts/${posts[0].id}`);

  return firstPostResponse.json();
}
```

> As it comes across an `await`, it wraps up the remainder of the code into a `callback` that waits for the promise to resolve -- passing control back to the caller.

```
async function getFirstPost() {
  const userResponse = await fetch("/users/1");
  const user = await userResponse.json();
  console.log(user);
}

const imaginaryCallbackOne = (userResponse) => {
  const user = await userResponse.json();
  console.log(user);
}

const imaginaryCallbackTwo = (user) => {
  console.log(user);
}
```

Question: What kind of object will be returned?

```
async function example2() {
  let str = "Hello World";
  console.log(str);
  return str;
}

const result = example2();
```

`async` functions always return a promise.

`await` can only be used within `async` functions.

Question: Anything wrong here?

```
async function getData() {
  const response = await fetch('/artists');
  return await response.json();
}
```

Exercise: Async & Await

1. Open `public/labs/ajax/index.js`
2. Implement the functionality
3. Test with jest:

   ```
   npm run test:watch public/labs/ajax
   ```

4. Test in your browser, if you want:

   localhost:3000/labs/ajax/

Bonus: Upgrade the previous `Promise` or `Fetch` labs to use `async/await`

Next Up

Forms

Demo at /demo/forms.html

Elements

- form
- input controls (input, select, textarea)
- submit controls

```
<form id="my-form" action="/save" method="POST" autocomplete="off">
  <input type="text" name="name" placeholder="Your Name">
  <input type="email" name="email" placeholder="Your Email">
  <input type="password" name="password">

  <input type="submit" value="Submit the thing">
</form>
```

Events

- form `submit`, `reset`
- input `change`, `focus`, `blur`, `input`
  - `change` occurs once typicall after losing focus
  - `input` as user types in certain fields

Validation

- disable with `novalidate` boolean `form` attribute
- some built-in options
  - `required` boolean attribute
  - `minlength` and `maxlength`
  - `type`
  - `pattern` for regex

```
<input id="choose" name="i_like" required pattern="[Bb]anana|[Cc]herry">
```

Form Values

- Properties on form elements
- Otherwise, using `FormData`

```
const input = document.querySelector('input[type=text]');
input.value;

// radio/checkboxes are a bit trickier
document.querySelector('input[name="rate"]:checked').value;
```

FormData

```
var formData = new FormData();

formData.append("username", "morris");
formData.append("id", 123456); // number 123456 is immediately converted to a string "123456"
```

```
const form = document.querySelector('form');
const data = new FormData(form);
```

Iterable

- .entries()
- .values()
- .keys()

```
const form = document.querySelector('form');
const data = new FormData(form);

for (let entry of data) {
    console.log(entry); // [fieldName, value]
}
```

Integrates nicely...

```
const userForm = document.querySelector('form.user');
const formData = new FormData(userForm);

fetch('/user', {
    method: "POST",
    body: formData
})
```

```
// or, if you want to build a query param string...
const queryString = new URLSearchParams(formData).toString();
// field1=val&field2=val&field3=val
```

Read more

Next Up

Web Sockets

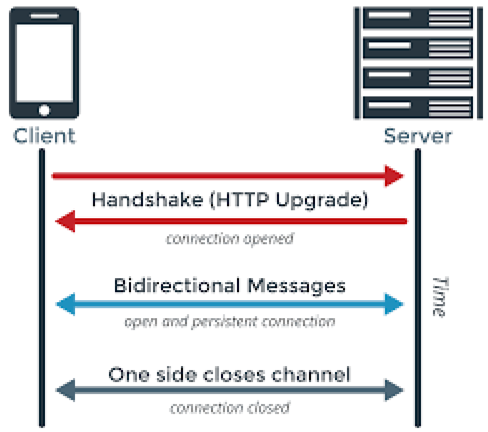
Web Sockets

- Full duplex connection (no more long polling)
- Operates over HTTP connection (upgraded to TCP/IP)
- Not subject to CORS
- No security restrictions; that's up to you
- Great for real-time data exchange: games, trading platforms, chat, etc

```javascript
// open a connection (ws or wss)
let ws = new WebSocket("wss://localhost:3030/");

// then listen for events
ws.onopen = function () {
  console.log("connected to WebSocket server");

  ws.send("Hello!");
};
```


Client
Server
Handshake (HTTP Upgrade)
connection opened
Bidirectional Messages
open and persistent connection
One side closes channel
connection closed
Time

WebSocket API

There are 4 events:

- `open`
- `close`
- `message`
- `error`

And two methods:

- `send(msg)` (string, Blob, ArrayBuffer)
- `close(code, reasonString)`

Sending

```
const message = {
  type: 'My Type',
  body: 'Anything I want to send',
  id: 5
};

ws.send(JSON.stringify(message));
```

Receiving

```
ws.onmessage = function (event) {
  const message = JSON.parse(event.data);

  console.log("incoming message: ", message);
};
```

Closing

```
socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Connection closed cleanly, code=${event.code} reason=${event.reason}`);
  } else {
    // e.g. server process killed or network down
    // event.code is usually 1006 in this case
    alert('[close] Connection died');
  }
};
```

Status codes

```
let specificStatusCodeMappings = {
    '1000': 'Normal Closure',
    '1001': 'Going Away',
    '1002': 'Protocol Error',
    '1003': 'Unsupported Data',
    '1004': '(For future)',
    '1005': 'No Status Received',
    '1006': 'Abnormal Closure',
    '1007': 'Invalid frame payload data',
    '1008': 'Policy Violation',
    '1009': 'Message too big',
    '1010': 'Missing Extension',
    '1011': 'Internal Error',
    '1012': 'Service Restart',
    '1013': 'Try Again Later',
    '1014': 'Bad Gateway',
    '1015': 'TLS Handshake'
};
```

Exercise: Chat Room

Create a simple chat interface to interact with the class:

1. Open `public/labs/chat-simple/index.js`
2. Implement it
3. Visit http://localhost:3000/labs/chat-simple/ to test

Our server: `wss://happy-family-chat-time.herokuapp.com/?token=`

TOKEN to be provided by Instuctor

Next Up

Web Storage

Storage APIS

- Allows you to store key/value pairs
- Two levels of persistence and sharing
- Very simple interface
- Keys and values must be strings

Session Storage

- Lifetime: same as the containing window/tab
- Sharing: Only code in the same window/tab
- 5MB user-changeable limit (10MB in IE)
- Basic API:

```
sessionStorage.setItem("key", "value");
let item = sessionStorage.getItem("key");
sessionStorage.removeItem("key");
```

Local Storage

- Lifetime: unlimited
- Sharing: Same domain
- 5MB user-changeable limit (10MB in IE)
- Basic API:

```
localStorage.setItem("key", "value");
let item = localStorage.getItem("key");
localStorage.removeItem("key");
```

The Storage Object

Properties and methods:

- length: The number of items in the store.
- key(n): Returns the name of the key in slot n.
- clear(): Remove all items in the storage object.
- getItem(key), setItem(key, value), removeItem(key)

Optional Exercise: Chat Room Upgrade

> Upgrade our chat room so that it stores the messages a user has seen/sent in the local storage.

When a user visits the page (or refreshes) load any previous messages from `localStorage` and render them.

1. Edit `public/labs/chat-simple/index.js`
2. Implement the upgrade
3. Visit http://localhost:3000/labs/chat-simple/ to test

Next Up

Location, URL and History

Location

`window.location` represents the current resource loaded by the user

```
window.location; // "https://google.com"

// change the page, stores in History
window.location.assign("https://another-page.com");

// force re-load the page
window.location.reload(true);

// does not store the new resource in History
window.location.replace("https://another-page.com");
```

Search Params

URLSearchParams provides an interface to manage url query parameters.

```
const params = new URLSearchParams('q=search+string&version=1&person=Eric');

params.get('q') === "search string"
params.get('version') === "1"
Array.from(params).length === 3
```

```
params.set('name', 'Jim');
params.delete('name');

params.set('names', 'Jim');
params.append('names', 'Jan');
params.getAll('names');

params.entries(); // iterable
params.forEach();

params.toString(); // "names=Jim&names=Jan"
```

The URL

We can parse a URL with the `URL` constructor.

```
const url = new URL('https://example.com?foo=1&bar=2');
const params = new URLSearchParams(url.search);
params.set('baz', 3);

params.has('baz') === true
params.toString() === 'foo=1&bar=2&baz=3'
```

`window.location` contains the current URL of your visitor.

```
// URL: https://example.com?version=1.0
const params = new URLSearchParams(location.search);
params.set('version', 2.0);

const url = new URL(window.location);
url.searchParams;
```

Other places URL Params come into play...

- You can pass `params` directly to `fetch`s `body` option
- Anchor tags get a `searchParams` property

```
const aLink = document.createElement('a');
aLink.href = "https://google.com?filter=api";

aLink.searchParams.get('filter'); // "api"
```

History

Interface to view and manipulate the current session's history.

Primarily allows a single-page/complex app to control the experience when a user clicks back or forwards, or copies the URL to share.

```
// arbitrary data can be stored in state
const state = { 'page_id': 1, 'user_id': 5 }
const title = ''
const url = 'hello-world.html'

history.pushState(state, title, url)
```

And then respond to history changes

```
window.addEventListener('popstate', (event) => {
  console.log("location: " + document.location + ", state: " + JSON.stringify(event.state));
});
history.pushState({page: 1}, "title 1", "?page=1");
history.pushState({page: 2}, "title 2", "?page=2");
history.replaceState({page: 3}, "title 3", "?page=3");

history.back(); // Logs "location: http://example.com/example.html?page=1, state: {"page":1}"
history.back(); // Logs "location: http://example.com/example.html, state: null"
history.go(2);  // Logs "location: http://example.com/example.html?page=3, state: {"page":3}"
```

Next Up

Functional (Revisited)

Core Concepts

- Higher Order Functions
- Pure Functions
- Immutable Variables
- Referential Transparency

Usually comes with (or enables):

- Partials
- Function composition
- Currying
- Point-free programming

Makes code

> Easier to reason about
>
> Reduces bugs
>
> Easier to test
>
> Faster to write
>
> Easier to test
>
> More flexible

Pure Functions

- Idempotency
- No side effects
- Can be run in parallel

```
// pure
function sum(x, y) {
    return x + y;
}

// not so pure - not idempotent
function sumIfSaturday(x, y) {
    const today = new Date();
    if (today.getDay() == 6) {
        return x + y;
    }
}
```

Pure / Impure

```
const user = {id: 5};

function augment(obj, name) {
    obj.name = name;
}
augment(user); // user is mutated
```

```
function augment(obj, name) {
    return Object.assign({}, obj, {name});
}
const augmentedUser = augment(user);
```

> We aren't forbidden from having side effects,
> but we do want to write "pure" where possible.

Immutable Variables

Help to maintain a state that is consistent and trustworthy.

- Variables are constant
- Objects should be `frozen`

```
let currentUser = {};
const status = {
    isLoggedIn: false,
};

function login(formData) {
    authenticate(formData).then((user) => {
        currentUser = user;
        status.isLoggedIn = true;
    });
}
```

What about `this`?

Objects and the prototype are not following a functional paradigm. `this` implies you're dependent on some external state or about to mutate it.

Objects can be functional (but you need to freeze them).

```
function makeUser(id, name) {
    return Object.assign(
        Object.create(null), // destroy `this`
        {
            id,
            name
        }
    );
}
const user = makeUser(5, "Tony Danza");

const adminUser = upgradeUser(user);
```

Referential Transparency

> Functions that can be replaced by their return value without impacting the execution of the application.

Functions that produce side effects are likely not referentially transparent.

`void` return values are a sign that the function is impure and not referentially transparent.

Partial Application

Create new functions with preloaded arguments.

```
function sum(...args) {
    return args.reduce((arg, accumulator) => {
        return arg + accumulator;
    }, 0);
}

const add100 = sum.bind(null, 100);
const remove30 = sum.bind(null, -30);
```

Currying

> Ability to call a function with fewer arguments than it expects, getting a new function that accepts the remaining arguments.

Enables breaking down functions into composable components.

```
const match = curry((what, s) => s.match(what));

match(/r/g, 'hello world'); // [ 'r' ]

const hasLetterR = match(/r/g); // x => x.match(/r/g)
hasLetterR('hello world'); // [ 'r' ]
```

```
function curry(fn) {
  const arity = fn.length;

  return function $curry(...args) {
    if (args.length < arity) {
      return $curry.bind(null, ...args);
    }

    return fn.call(null, ...args);
  };
}
```

Credit

Read Up / Be more functional

- Libraries like Ramda and Lodash
- Mostly Adequate Guide

Next Up

Tooling

Modern Web Dev requires a lot of things...

- Not all ECMAScript can run in every browser
- ES Module management
- Package Dependency management, tree shaking
- Compile SASS/SCSS -> CSS
- Optimize images and fonts
- Generate the right HTML (from templates?)
- Minify our files
- Automating tests? Why not!
- Code quality? Why not!

> JavaScript can be leveraged to do all these things

- Tasks
- Transpilers
- Bundlers
- Task Runners

Linting / Standardized Format

- PrettierJS
- ESLint

Transpile

- Babel

Bundle and deploy

- Parcel
- Brunch
- Webpack

Combine them all with runners

- NPM / Yarn
- Webpack (and Parcel, and brunch)
- Grunt
- Gulp (Dead?)
- Rollup
- Always a new one...

Demos?

- Set up a Linter: Prettier or ESLint
- Set up babel
- Set up a bundler/build tool: Parcel

Next Up

Testing JavaScript with Jest

Jest

- Jest
- Based off Jasmine
- Unit tests (versus integration)
- Runs from CLI (via node)
- Interesting `snapshot` test system for components
- All tests run in parallel, isolated from one another
- Built in code coverage, assertion libs, mock library

Example:

```
import {sum} from './index.js';

test("sum x and y", () => {
  expect(sum(3, 5)).toBe(8);
});

it("should sum x and y", () => {
  expect(sum(3, 5)).toBe(8);
});
```

Set up

- Install via node/yarn
- Configure with `npx jest --init`
- Uses CommonJS modules out of the box :/
- To use ES Modules:
    i. passing experimental modules node flag
    ii. Add `"type": "module"` to your `package.json`
    iii. Disable transforms in your jest config: `transform: {}`

Running it

```
npx jest
npx jest --watch
npx jest --watchAll

npx jest /some/path/

# only run against changed files
npx jest -o
```

Basic Expectation Matchers

```
expect(resultValue).toBe(expectedValue);
```

- `toBe(x)`: Exact comparison, using `Objext.is`
- `toEqual(x)`: Equality, including deep property comparison
- `.not.` modifier to negate a matcher
- `toContain(x)`: Check iterable elements

- `toBeDefined()`: Confirms expectation is not undefined.
- `toBeUndefined()`: Opposite of toBeDefined().
- `toBeNull()`: Confirms expectation is null.
- `toBeTruthy()`: Should be true true when cast to a Boolean.
- `toBeFalsy()`: Should be false when cast to a Boolean.

Numeric Expectation Matchers

- `toBeLessThan(n)`: Should be less than n.
- `toBeGreaterThan(n)`: Should be greater than n.
- `toBeGreaterThanOrEqual(n)`: Should be greater than n.
- `toBeCloseTo(e, p)`: Difference within p places of precision.

Strings Matchers

- `toMatch(regex)`

Exceptions

- `toThrow()`
- `toThrow(Error)`: Throw a specific Error
- `toThrow(string)`: Check error message

The basic unit

```
describe("Module", () => {
  test("Something happens", () => {
    expect(/* */);
  });

  it("Should do something", () => {
    expect(/* */);
  });
});
```

Exercise: Basic Unit Test

1. Open `public/labs/jest/adder.spec.js`
2. Read the code then do exercise 1 (we'll do exercise 2 later)
3. Execute the test framework:

```
npm run test:watch public/labs/jest
```

Life Cycle Callbacks

Each of the following functions takes a callback as an argument:

- `beforeEach`: Before each it is executed.
- `beforeAll`: Once before any it is executed.
- `afterEach`: After each it is executed.
- `afterAll`: After all it specs are executed.

```
beforeEach(() => {
  initializeCityDatabase();
});

afterEach(() => {
  clearCityDatabase();
});
```

Only run one unit

```
test.only('I want to only run this one', () => {
  // stuff...
});

test('I will not be run', () => {});
```

Mocking

You can fake, spy on and replace implementations of code to write isolated unit tests.

```
const mockFn = jest.fn();

mockFn("Hello");
mockFn("Mock World");

expect(mockFn).toHaveBeenCalled();
expect(mockFn).toHaveBeenCalledTimes(0);
expect(mockFn.mock.calls.length).toBe(2);
expect(mockFn.mock.calls[0][0]).toBe("Hello"));

// With a mock implementation:
const returnsTrue = jest.fn(() => true);
console.log(returnsTrue()); // true;
```

Return Values

```
const myMockCallback = jest.fn();

myMockCallback
  .mockReturnValueOnce(10)
  .mockReturnValueOnce('x')
  .mockReturnValue(true);
```

Modules

```
jest.mock('axios');

test('should fetch users', () => {
  const users = [{name: 'Bob'}];
  const resp = {data: users};
  axios.get.mockResolvedValue(resp);

  // or you could use the following depending on your use case:
  // axios.get.mockImplementation(() => Promise.resolve(resp))

  return Users.all().then(data => expect(data).toEqual(users));
});
```

Spying (Call Counting)

> `spyOn` will create a mock `fn` and also track calls on object methods

```
const foo = {plusOne: n => n + 1};

test("should be called", function () {
  jest.spyOn(foo, "plusOne");

  let x = foo.plusOne(42);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(foo.plusOne).toHaveBeenCalledTimes(1);
  expect(foo.plusOne).toHaveBeenCalledWith(42);

  expect(x).toBeUndefined();
});
```

Spying and Calling Through?

Jest automatically does this with spies; you can disable this or fake it:

```
test("should call through and execute", function () {
  jest.spyOn(foo, "plusOne").mockImplementation(() => {
    console.log("I am fake");
  })

  let x = foo.plusOne(42);

  expect(foo.plusOne).toHaveBeenCalled();
  expect(x).toBe(43);
});
```

Clear all mocks

```
// unmocks all
jest.clearAllMocks()

// keep the mocks, but reset their state
jest.resetAllMocks()
```

Exercise: Using Mocks

1. Open `public/labs/jasmine/adder.spec.js`
2. Read the code then do exercise 2
3. Execute the test framework:

   ```
   npm run test:watch public/labs/jest
   ```

Mocking the DOM

- Jest comes with `js-dom`
- Replicates the DOM in Node for our tests

```
it("should manipulate the dom", () => {
  document.body.innerHTML = `<div>Nothing to see here, folks</div>`;
  expect(document.querySelector('div').length).toBe(1);
});
```

Or see the `find the flags` tests

Mocking Events

- Test the handler in isolation when possible
- You can mock `document.addEventListener` to track handlers added
- Using a library like `fireEvent` or `user-event`

```
it("lib render should set up a handler", () => {
  const mock = jest.fn();
  document.body.innerHTML = `<div>Nothing to see here, folks</div>`;

  lib.render();

  const anchorTag = document.querySelector("a.lib-link");
  const event = new MouseEvent("click");
  anchorTag.dispatchEvent(event);

  expect(mock).toHaveBeenCalledTimes(1);
});
```

Exercise: DOM Testing

1. Open `public/labs/jest/adder.spec.js`
2. Read the code then do exercise 3 and 4 (optional)
3. Execute the test framework:

   ```
   npm run test:watch public/labs/jest
   ```

Mocking Fetch

Use a lib or do it yourself.

```
global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ users: [1, 2, 3]}),
  })
);

beforeEach(() => {
  fetch.mockClear();
});
```

Testing Time-Based Logic (Setup)

```
jest.useFakeTimers();

const runLater = (fn) => setTimeout(fn, 1000);

test("Timers", () => {
  const callback = jest.fn();

  runLater(callback);

  expect(callback).toHaveBeenCalledTimes(0);

  jest.advanceTimersByTime(1001);
  expect(callback).toHaveBeenCalledTimes(1);
});
```

Testing Asynchronous Functions

```
test("uses an asynchronous function", function (done) {
  // `setTimeout' returns immediately,
  // so this test does too!
  setTimeout(function () {
    expect(done instanceof Function).toBeTruthy();
    done(); // <-- tell Jasmine are are all done
  }, 1000);
});
```

Testing Promises

> Use `then` and `done`

```
// testing `apiGet`, which returns a promise...
test("a promise", function (done) {
  const apiGetterResult = apiGet('/some/data');

  apiGetterResult.then((result) => {
    expect(result).toEqual({id: 1});
    done();
  })
});
```

Testing Promises

> Return the promise

```
test("a promise", function () { // <-- no more "done"!
  const apiGetterResult = apiGet('/some/data');

  return apiGetterResult.then((result) => {
    expect(result).toEqual({id: 1});
  })
});
```

Testing Promises

> Use `resolves` or `rejects`

```
test("a promise", function () {
  const apiGetterResult = apiGet('/some/data');

  // return the promise! otherwise use `done`
  return expect(apiGetterResult).resolves.toEqual({id: 1});
});
```

Async/Await

```
test("some async code", async function () {
   // "We expect 1 assertion here..."
   // not necessary but good practice
  expect.assertions(1);

  const result = await apiGet('/some/data');
  expect(result).toEqual({id: 1});
});
```

Exercise: Asynchronous Testing

1. Open `public/labs/jest/delayed.spec.js`
2. Read the code then do exercise 5 and the bonus
3. Execute the test framework:

   ```
   npm run test:watch public/labs/jest
   ```

Next Up

Web Components

Web APIS are evolving

> ...paving the cow paths

- Browsers have become more consistent
- UI Components are a common pattern
- More control being given to the developer

Web Components as a UI Component

Made up of a collection of browser APIs for creating *reusable*, *encapsulated* and *exstensible* HTML elements

```html
<rainbow-button>Click me!</rainbow-button>

<custom-modal trigger=".modal-triggers">
  <h1>Welcome to my site</h1>
  <signup-form source="welcome-modal"></signup-form>
</custom-modal>
```

See Demo: http://localhost:3000/demo/web-components/demo-1.html

React Components === Web Components?

> React and Web Components are built to solve different problems. Web Components provide strong encapsulation for reusable components, while React provides a declarative library that keeps the DOM in sync with your data.
>
> The two goals are complementary. As a developer, you are free to use React in your Web Components, or to use Web Components in React, or both.

Source

Why Web Components

Standardizes & encapsulates JavaScript-enhanced UI components without a framework.

- Interoperability
- Lifespan
- Portability

Drawbacks

- Still see minimal support
  - polyfills for up to v1 web components
- Requires JS, no fallbacks
- Challenging to let them degrade gracefully
- Not a full framework

Web Component APIs

- Custom Elements
- Shadow DOM
- HTML Templates

Custom Elements API

Register custom elements and their behavior (through a `class`).

```
// define the behavior in a class - it should extend HTMLElement
class RainbowButton extends HTMLElement {
  constructor() {
    super(); // super() must come first...
    this.textContent = "Rainbow";
  }
}
// register the tag - it must include a hyphen
customElements.define("rainbow-button", RainbowButton);
```

Then you can use that element

```html
<!-- direclty in your HTML -->
<rainbow-button></rainbow-button>
```

```javascript
// or create it on the fly (*requires shadow dom)
const newButton = document.createElement("rainbow-button");

// or instantiate an instance
const otherButton = new RainbowButton();
```

Question: Spot any pitfalls here?

```
<rainbow-button>click me</rainbow-button>
```

```
class RainbowButton extends HTMLElement {
  constructor() {
    super();
    const p = document.createElement("p");
    p.textContent = `🌈${this.textContent}🌈`;
    this.appendChild(p);
  }
}

// elsewhere...
document.querySelectorAll("p").forEach((el) => {
  el.style.backgroundColor = "purple";
});
```

Question: Will these styles affect the `<p>` in our web component so far?

```
<style>
p {
  color: red;
}
</style>
```

Shadow DOM vs Light DOM

- Elements can have their own DOM tree
  - Private and hidden
- Isolates custom elements
  - CSS (mostly)
  - DOM (ie: js selection)
  - Retargets bubbled events
- Can be *visible* or *hidden*

See Demo: http://localhost:3000/demo/web-components/demo-1-shadow.html

Using the Shadow DOM

```
class RainbowButton extends HTMLElement {
  constructor() {
    super();

    // this can be "open" or "closed"
    // "closed" will require you store a reference to the shadow root
    this.attachShadow({ mode: "open" });

    const el = document.createElement("p");
    el.textContent = "Rainbow";

    this.shadowRoot.appendChild(el);
  }
}
```

...*What did this do?*

Inspecting the `<rainbow-button>` element will reveal a `#shadow-root`

```
// won't select <p> tags inside <rainbow-button> elements
document.querySelectorAll("p").forEach((el) => {
  el.style.backgroundColor = "purple";
});
```

```
/* won't affect inner elements of <rainbow-button> elements */
p {
  color: red;
}
```

Exercise

Implement a basic custom element

Edit: `public/labs/web-components/exercise-1/index.html`

Test: http://localhost:3000/labs/web-components/exercise-1/index.html

Event handling in components

- Events are **re-targeted**
- Most will bubble through a Shadow DOM boundary
  - exceptions: mouseenter, load, select, slotchange
  - custom events need `composed:true`
- `e.composedPath()` reveals full path of an event

However, *slotted* element handlers **do** start from the light dom element...

Question: What will the `target` be in these two event handlers? (*see:* `/demo-1-events.html`)

```
class RainbowButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
    const el = document.createElement("p");
    el.textContent = "Rainbow";
    this.shadowRoot.appendChild(el);

    el.addEventListener("click", (e) => {
      console.log(e.target); // here?
    });
  }
}

document.querySelectorAll("rainbow-button").forEach((el) => {
  el.addEventListener("click", (e) => console.log(e.target)); // here?
});
```

HTML Templates

Browser-native way to create reusable HTML templates.

- Doesn't get "parsed"
- Won't render
- Won't count against node count

```
<template id="rainbow-button-template">
  <h1>My Rainbow Button</h1>
</template>
```

Template Elements

- Have a `.content` property with a `DocumentFragment`
- We can copy that with `cloneNode`

```html
<template id="rainbow-button-template">
  <h1>My Rainbow Button</h1>
</template>
```

```javascript
const tmpl = document.getElementById("rainbow-button-template");
console.log(tmpl.content); // # DocumentFragment "<h1>My Rainbow Button</h1>"

const newElFromTemplate = tmpl.content.cloneNode(true);
```

Tying it together

```
class RainbowButton extends HTMLElement {
  constructor() {
    super();

    this.attachShadow({ mode: "open" });

    const template = document.getElementById("rainbow-button-template");
    const templatedElement = template.content.cloneNode(true);

    this.shadowRoot.appendChild(templatedElement);
  }
}
```

Template Slots

- Works with `<template>` tags
- Insert light dom elements into a shadow dom
- **Live** reference

Template slots in action

```html
<template id="rainbow-button-template">
  <h1>My Rainbow Button</h1>
  <button>
    <!-- this <slot> will hold the light dom element given by the user -->
    <slot name="button-label">TBD</slot>
  </button>
</template>
```

```html
<rainbow-button>
  <!-- this <span> will be referenced inside the <slot> -->
  <span slot="button-label">Click for Magic</slot>
</rainbow-button>
```

End result

```
<rainbow-button>
  <!-- this is the <span> I provided -->
  <span slot="button-label">Click for Magic</slot>
  <!-- this is the rendered shadow root -->
  #shadowroot
    <h1> My Rainbow Button</h1>
    <button>
      <slot name="label">
        TBD
        <!-- "<span slot="button-label">Click for Magic</slot>" is referenced here -->
      </slot>
    </button>
</rainbow-button>
```

Let's check out a demo: demo-1-template.html

Question: If I were to stick a form in the **slotted element** after the page has rendered, what do you expect to see happen?

```
document.querySelector('span[slot="button-label"]').innerHTML = `
<form><input type="text"></form>
`;
```

```
<rainbow-button>
  <span slot="button-label">Click for Magic</slot>
</rainbow-button>
```

Thinking about templates

- Define your structure
  - You can programmatically create a `<template>`
- Give control to the user (optional)
- Accept light dom elements

Lifecycle callbacks

Run code only once a custom element is connected to the DOM -- **good for expensive operations** or avoiding unnecessary requests prior to insertion.

```
class RainbowButton extends HTMLElement {
  connectedCallback() {
    // each time this element is appended into a document-conected tree
    // ex: .appendChild(myComponentInstance);
  }

  disconnectedCallback() {
    // each time this element is removed from a document-connected tree
    // ex: parentNode.removeChild(myComponentInstance)
  }
}
```

Take note: these may run multiple times for the same node.

Working with attributes

```
class RainbowButton extends HTMLElement {
  static get observedAttributes() {
    return ["button-type", "class"];
  }

  attributeChangedCallback(attrName, oldValue, newValue) {
    // each time one of the watched attributes change
  }
}
```

Cascading Styles

CSS is scoped within the **shadow root**; inner styles don't leak out and *most* outer styles won't leak in.

- Some styles still cascade in...
  - background, color, font, line-height, etc...
- CSS Vars **are** passed through
- Slotted elements are styled by the main page only

More with Web Components

- Extending built-ins (like Buttons, Forms, etc)
- Styling
- Events (Retargeted)

Best Practices w/ Web Components

- Do use a Shadow Dom to encapsulate
- Do pass primitive data as attributes (not nodes)
- Do dispatch events based on internal activity
- Don't hijack the `class` attribute
- Don't do expensive operations in your constructor
- Avoid registering handlers in your constructor

Exercise

Improve this custom element with a template & slot.

Edit: `public/labs/web-components/exercise-2/index.html`

Test: http://localhost:3000/labs/web-components/exercise-2/index.html

Next Up

Common Attack Vectors

- XSS
- CSRF

Cross Site Request Forgery

- Attacker tricks user to click a link or visit a page
- Link/page is a request to a friendly resource (ex: a bank)
- That triggers some action on behalf of the user (ex: transfer money)
- Prevent by same-origin / strict CORS policy

  This is bad: `Access-Control-Allow-Origin: *`

```html
<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0" height="0" border="0">
```

```html
<script>
function send() {
    var x = new XMLHttpRequest();
    x.open("PUT","http://bank.com/transfer.do",true);
    x.setRequestHeader("Content-Type", "application/json");
    x.send(JSON.stringify({"acct":"BOB", "amount":100}));
}
</script>

<body onload="send()">
```

Cross Site Scripting (XSS)

- Get the site to inject malicious javascript
- Allowing attacker to act as the user
- Can gain access to the user data and credentials
- And potentially more...

Reflected XSS

- When raw markup is taken from the query parameters
- Server and client-side should be sanitizing it

```
// if a site allows this:
https://insecure-website.com/status?message=All+is+well.


// then we can attack it:
https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script><p>Status: <script>/- Bad stuff here... */</script></p>
```

Stored XSS

- Similar to before, but the markup is stored in the database via user submission or request to a 3rd party API
- Server should be sanitizing content before storing it
- Also good practice to sanitize prior to render

```
<textarea name="comment">
    <script>/*+Bad+stuff+here...+*/</script>
    <p>This is my comment</p>
</textarea>
```

DOM Based XSS

- When some raw user input is rendered elsewhere on the page
- If an attacker can hijack that input (ex: via query param) they can execute a script

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

```
<input name="search">
<p>Searched for: <%= search %></p>
```

Avoiding these attacks

- Filter input
- Encode output
- Set appropriate headers to ensure you send/get the right content
- Use a CSP

Content Security Policy

- `Content-Security-Policy` response header that indicates a policy (or a `meta` tag)
- Can restrict to a domain, page, hash (of the resource), key
- A series of directives

```
# prevent XSS
# allow scripts from only the "same origin" as self
script-src 'self'

# allow from the domain
script-src https://scripts.normal-website.com

# images
# prevents "dangling markup" attacks
img-src 'self'

# prevent iframes around me from unknown sources
# prevents "click jacking"
frame-ancestors 'none'
```

Next Up

That's a wrap!

- Web Workers?
- Memory Management?
- Multithreaded?
- Generators (Promise Generators/Async Generators)
- More tooling

Final Exercise - Artists Component

> Create a web component that renders an artist in the page, given an artist `id`. The artists will be fetched from the local api.

- Work in `public/labs/artists-component`
- Implement it!
- Test it (write tests if you like):

  `http://localhost:3000/labs/artists-component/`