

Intermediate JavaScript

Introductions

- What is your experience with JavaScript?
- Why are you taking the class?

Our Goals

To get *really good* at the fundamentals

- Hoisting
- Coercion
- Scope
- Context
- The prototype

+ We'll also get decent exposure to some modern JS and browser APIs

Outline

- The fundamentals
- Functions
- Objects (OO / Prototype / Classes)
- The DOM
- Handling events
- Promises
- Fetch (Ajax)
- Web Components
- Testing JS (Jasmine)

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

The Fundamentals

Coercion

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

```
8 * null; // 0
```

```
null > 0; // false
```

```
null === 0; // false
```

```
null >= 0; // true ???
```

```
[] + []; // ""
```

```
[] - []; // 0
```

```
+"5"; // 5 <-- it converted it for me
```

```
!!val; // coerces to a boolean
```


Equality

- With or without *coercion*
- A common cause of bugs & confusion

```
// loose
"1" == 1; // true
[3] == "3"; // true
[3] == 3; // true

// strict
"1" === 1; // false
[3] === "3"; // false

// most strict (ES6+)
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
// prettier-ignore
{} // Empty object
Infinity; // Yep, it's true
```

Question: Is there anything wrong/misleading about the following?

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

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

Scope

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

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

Question: What is the scope of `x` ?

```
var x = 10;

function hello() {
  // which `x` is being accessed?
  // what if we put a var in front of this x?
  x = 1;
}

hello();
```

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

```
let z = 12;  
  
if (z > 0) {  
  let x = 10;  
  
  if (x < 100) {  
    x = x + z;  
  }  
}
```

Variables in *outer* scopes can be accessed from *inner* scopes

Question: Will this be OK?

```
let rad = 12;

if (rad > 0) {
  let amazing = 10;

  if (amazing < 100) {
    amazing++;
  }
}

let result = amazing + rad;
```

Outer scopes cannot access variables in inner scopes

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;  
}
```

Block vs Function scope

`var` declares a variable within a function's scope.

`let` and `const` declare within a block's scope.

There are other nice things about `let` and `const` ...

Question: Will any lines produce an error here?

```
const x = 5;  
const y;  
const apple = { color: "red" };  
  
x = 12;  
apple.color = "blue";
```

const will create 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

- Not all variables are created equally
- `var` 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 and Block scope
- `var` is hoisted to the top of its scope

Before we get to our first lab...

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

Exercise: Scope

This is just 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

- Did you use any `const` or `vars` ?
- What are some pitfalls with this code?
- Is it "functional"?
- Would it be easy to test?
- How can we improve it... modules?

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 recursor = 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

```
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
```

Question: What will this do?

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


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 `...` 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
```

Destructuring in function parameters

```
function updateUser({ id, name, isFriendly = false }) {  
  console.log(`I am going to update user ${id}`);  
}  
  
const user = {  
  id: 5,  
  name: "Tigger",  
  isFriendly: true,  
};  
  
updateUser(user);
```

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

In this way we can break down our code into reusable, easy to test function components.

Go from this:

```
const names = ["abe", "bob", "carol"];
let allNames = "";

for (let i = 0; i < names.length; i++) {
  allNames += ` ${names[i]} `;
}
```

To something more functional:

```
// now we can move this into a lib/module...
const nameReducer = (acc, name) => {
  return `${acc} ${name}`;
};
let allNames = names.reduce(nameReducer);
```


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

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

let even = numbers.filter(function (n) {
  return n % 2 === 0;
});

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

Mapping over an array

```
let strings = [  
  "Mon, 14 Aug 2006 02:34:56 GMT",  
  "Thu, 05 Jul 2018 22:09:06 GMT",  
];  
  
let dates = strings.map(function (s) {  
  return new Date(s);  
});  
  
dates; // [Date, Date]
```

Reducing an array

```
let a = [1, 2, 3];  
  
// Sum numbers in `a`.  
let sum = a.reduce(function (acc, elm) {  
    return acc + elm;  
}, 0); // initial value of accumulator  
  
sum; // 6
```

Exercise: Arrays & Functional Programming

1. Open the following file:

```
public/labs/array/array.js
```

2. Complete the exercise.

3. Run the tests by visiting <http://localhost:3000/labs/array>

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

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!");
```



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

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

```
// 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!");
```

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
- Old world modules
- Maintain state

```
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/closure.js`
2. Do the exercise to pass all tests
3. To test and debug, open
<http://localhost:3000/labs/closure/>

Modules

Looking back at our earlier exercise

```
function MyModule() {  
  let callCount = 0;  
  let name;  
  
  function promptUserForName() {  
    callCount++;  
  
    if (callCount > 1) {  
      return;  
    }  
  
    name = prompt("What is your name?");  
  }  
  
  function getCurrentName() {  
    alert(`The last name was: ${name}`);  
  }  
  
  return {  
    promptUserForName,  
    getCurrentName,  
  };  
}
```

We can take it one step further...

```
const MyModule = (function () {  
  let callCount = 0;  
  let name;  
  
  function promptUserForName() {  
    callCount++;  
  
    if (callCount > 1) {  
      return;  
    }  
  
    name = prompt("What is your name?");  
  }  
  
  function getCurrentName() {  
    alert(`The last name was: ${name}`);  
  }  
  
  return {  
    promptUserForName,  
    getCurrentName,  
  };  
})();
```


The IIFE

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

Modules

Encapsulated, easy to understand and maintain, well-organized, protected

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

Examples can be seen in `public/solutions/scope`

Exercise: Hosts

But first, let's look at the [Map object](#)...

1. Open `public/labs/hosts/hosts.js`
2. Do the exercise to pass all tests
3. To test and debug, open
<http://localhost:3000/labs/hosts/>

ES Modules

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

```
// index.js  
import { util } from "./myLib.js";  
import emptyFunction from "./myLib.js";
```

ES Modules

- It's a live binding
- `export` and `import`
- named exports or a default
- [Not totally supported](#) (but almost)

I have a [demo](#)

Post exercise...

Can we convert `hosts` into a modern es module?

Sure can...

Objects and Context

Question: Why is this `sayHello` a bummer?

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


Context

`this` references the context of a function

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

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

```
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;
```

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

Consider:

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

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);  
  },  
};
```

OO in JS

Creating Objects

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

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 Functions

1. Open the following file: `public/labs/constructors/constructors.js`
2. Complete the exercise.
3. Run the tests by opening <http://localhost:3000/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();
```

More with classes

```
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.
```

Exercise: Class Upgrade

1. Revisit `public/labs/constructors/constructors.js`
2. Upgrade your Constructor function to use a `class` instead
3. Run the tests by opening <http://localhost:3000/labs/constructors/>

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

HTML Elements

```
<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?

```
#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>
```

The DOM

- What most people hate(d) in the browser
- The Browser's API for the document
- Represents elements as a tree of nodes
- Live data structure

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

Element Nodes

The HTML:

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

Maps *loosely* to:

```
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

There are performance considerations when it comes to modifying the DOM.

Selecting

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

```
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

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

```
let el = document.querySelector(".the-child");  
  
el.children[0].innerHTML = "<h1>Hi!</h1>";  
el.parentElement;
```

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

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

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

DataSet API

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

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

2. Visit <http://localhost:3000/labs/flags>.

3. Complete the exercise.

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 and trigger registered handler functions
 - click, page ready, focus, submit, scroll, etc...
- Browser implements an event loop to process handlers
 - one function at a time; it is blocking

Demo a Runtime: </demo/runtime/>

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>

Handling Events

- Select an element
- Define a handler function
- Register the handler on the element

```
const myFunction = function () {};  
  
const el = document.getElementById("container");  
  
el.addEventListener("click", myFunction);
```

Handler Functions

- Always passed an "event object" by the browser
- `context` is the element where the handler is registered
- You can de-register them

```
const myFunction = function (eventObject) {  
  console.log(this); // element where I am registered  
  
  eventObject.target; // same as ^  
  eventObject.currentTarget; // element that is currently handling the event...  
};
```

Event Propagation

- Events move throughout the entire DOM tree (from the source of the event to the top level dom node)
- Trickles (first) then Bubbles (second)
- You can control it!

```
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

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

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();
```

A full event handler example

```
node.addEventListener("click", function (event) {  
  // `this` === Node the handler was registered on.  
  console.log(this);  
  
  // `event.target` === Node that triggered the event.  
  console.log(event.target);  
  
  // Add a CSS class:  
  event.target.classList.add("was-clicked");  
  
  // You can stop default browser behavior:  
  event.preventDefault();  
});
```

Exercise: Simple User Interaction

1. Open the following files in your text editor:

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

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

Loading data / 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 old way of doing AJAX
- Inconsistent and 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 (but Edge is all good)

```
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);
```

Promises

- Standardized construct to represent some future data
- Composable
- Three states: Pending, Fulfilled, Rejected
- Flattens asynchronous code that would otherwise be deeply nested

This old callback pyramid...

```
// this is a rough sketch of 3 ajax requests, each dependent on the previous
req.open("GET", "/users/1.json");

req.addEventListener("load", () => {
  req2.open("GET", "/users/1/posts.json");

  req2.addEventListener("load", () => {
    req3.open("GET", "/posts/35.json");

    req3.addEventListener("load", () => {
      // got all our data!
    });
  });
});
```

Becomes more like:

```
fetch("/users/1.json")
  .then((d) => {
    return fetch("/users/1/posts.json");
  })
  .then((d) => {
    return fetch("/posts/35.json");
  });
```


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);  
  });  
};
```

then there is the promise consumer...

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);  
  
someOtherThingThatWorksPromises(prom);
```

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]);
```

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);
```

Exercise: Using the Fetch API

1. Start your server if it isn't running

2. Open `public/labs/fetch/fetch.js`

3. Fill in the missing pieces

The API is available at <http://localhost:3000/artists>

4. To test and debug, open

localhost:3000/labs/fetch/

Async & Await

Helps to unwrap "promises" to make asynchronous code read more synchronously

Take 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();
    }
  });
```

Make it 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();  
}
```


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.

Exercise: Async & Await

1. Start your server if it isn't running
2. Open `public/labs/ajax/ajax.js`
3. Fill in the missing pieces

The API is available at <http://localhost:3000/artists>

4. To test and debug, open
localhost:3000/labs/ajax/

Web Components

Web APIs are evolving

- Browsers have become more consistent
 - ...paving the cow paths
- 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 *extensible* HTML elements

```
<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

```
<!-- directly in your HTML -->  
<rainbow-button></rainbow-button>
```

```
// or create it on the fly (*requires shadow dom)  
const newButton = document.createElement("rainbow-button");  
  
// or instantiate an instance  
const otherButton = new RainbowButton();
```

Quiz

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";  
});
```

Quiz

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;
}
```

Quiz

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?
});
```

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...

Exercise

Implement a basic custom element

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

Test: <http://localhost:3030/labs/web-components/exercise-1.html>

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`

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

```
const tmp1 = document.getElementById("rainbow-button-template");  
console.log(tmp1.content); // # DocumentFragment "<h1>My Rainbow Button</h1>"  
  
const newElFromTemplate = tmp1.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

```
<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>
```

```
<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](#)

Quiz

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-connected 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.html`

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

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)`

Web Sockets

Web Sockets

- Full duplex connection (no more long polling)
- Not subject to CORS
- No security restrictions; that's up to you
- Operates over HTTP connection (upgraded to TCP/IP)

```
let ws = new WebSocket("ws://localhost:3030/");
ws.onopen = function () {
  console.log("connected to WebSocket server");

  ws.send("Hello!");
};
ws.onmessage = function (e) {
  console.log("incoming message: " + e.data);
};
```

Exercise: Simple Chat

Create a simple chat interface to interact with the class:

1. Open `public/labs/chat-simple/index.js`
2. Implement a web socket connection so that you can send/receive messages
3. Visit <http://localhost:3000/labs/chat-simple/> to test

You can use either `ws://localhost:3030` or `ws://happy-family-chat-time.herokuapp.com/?token=`

TOKEN to be provided by Instuctor

Testing JavaScript

Jasmine

- We'll use [Jasmine](#)
- Jest is based off Jasmine
- Spec-based testing
- Expectations instead of assertions

Example:

```
describe("ES2015 String Methods", function () {  
  describe("Prototype Methods", function () {  
    it("has a find method", function () {  
      expect("foo".find).toBeDefined();  
    });  
  });  
});
```

Basic Expectation Matchers

- `toBe(x)` : Compares x using `===` .
- `toMatch(/hello/)` : Tests against regular expressions or strings.
- `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.
- `toBeCloseTo(e, p)` : Difference within p places of precision.

Value Matchers

- `toEqual(x)` : Can test object and array equality.
- `toContain(x)` : Expect an array to contain x as an element.

Exercise: Writing a Test with Jasmine

1. Open `public/labs/jasmine/adder.spec.js`
2. Read the code then do exercise 1 (we'll do exercise 2 later)
3. To test visit [<http://localhost:3000/labs/jasmine/>]

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.

Spying

Given this set up code...

```
let foo;  
  
beforeEach(function () {  
  foo = {  
    plusOne: function (n) {  
      return n + 1;  
    },  
  };  
});
```

Spying (Call Counting)

```
it("should be called", function () {  
    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

```
it("should call through and execute", function () {  
    spyOn(foo, "plusOne").and.callThrough();  
  
    let x = foo.plusOne(42);  
  
    expect(foo.plusOne).toHaveBeenCalled();  
    expect(x).toBe(43);  
});
```

Spying and Calling a Fake

```
it("should call a fake implementation", function () {  
  spyOn(foo, "plusOne").and.callFake((n) => n + 2);  
  
  let x = foo.plusOne(42);  
  
  expect(foo.plusOne).toHaveBeenCalled();  
  expect(x).toBe(44);  
});
```

Exercise: Using Jasmine Spies

1. Open `public/labs/jasmine/adder.spec.js`
2. Read the code then do exercise 2
3. To test visit [<http://localhost:3000/labs/jasmine/>]

Testing Time-Based Logic (Setup)

```
let timedFunction;

beforeEach(function () {
  timedFunction = jasmine.createSpy("timedFunction");
  jasmine.clock().install();
});

afterEach(function () {
  jasmine.clock().uninstall();
});
```

Time-based Logic (setTimeout)

```
it("function that uses setTimeout", function () {  
    inFiveSeconds(timedFunction);  
  
    // The callback shouldn't have been called yet:  
    expect(timedFunction).not.toHaveBeenCalled();  
  
    // Move the clock forward and trigger timeout:  
    jasmine.clock().tick(5001);  
  
    // Now it's been called:  
    expect(timedFunction).toHaveBeenCalled();  
});
```

Time-based Logic (setInterval)

```
it("function that uses setInterval", function () {  
    everyFiveSeconds(timedFunction);  
  
    // The callback shouldn't have been called yet:  
    expect(timedFunction).not.toHaveBeenCalled();  
  
    // Move the clock forward a bunch of times:  
    for (let i = 0; i < 10; ++i) {  
        jasmine.clock().tick(5001);  
    }  
  
    // It should have been called 10 times:  
    expect(timedFunction.calls.count()).toEqual(10);  
});
```

Testing Asynchronous Functions

```
describe("asynchronous function testing", function () {  
  it("uses an asynchronous function", function (done) {  
    // `setTimeout` returns immediately,  
    // so this test does too!  
    setTimeout(function () {  
      expect(done instanceof Function).toBeTruthy();  
      done(); // tell Jasmine we were called.  
    }, 1000);  
  });  
});
```

Exercise: Asynchronous Testing

1. Open `public/labs/jasmine/delayed.spec.js`
2. Read the code then do exercise 3
3. To test visit [<http://localhost:3000/labs/jasmine/>]

And beyond!

- Web Workers
- Modernizing every year
- Tooling & Build systems

Resources

Get more

- [You Don't Know JS](#)
- <https://javascript.info/>
- [Mozilla](#)