JavaScript Fundamentals

Introductions

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

Setup

- Pre-reqs: Node & NPM
- Download github.com/rm-training/webdev
  - Slides are at: `/slides/fundamental-js.pdf`
- Download dependencies

```
npm install
```

- Start the server

```
npm start
```

- Visit `localhost:3000` and bask in the glory!

Our topics

- Language Basics
- Debugging & Tooling
- Scope & Hoisting
- Intro to OO in JS
- Exception handling
- Manipulating Web Pages
- Event Handling
- AJAX Requests
- Testing

Going further (not covered here):

- Deeper on Object Construction & Classes
- Function patterns
- Modules & the IIFE pattern
- Controlling context w/ call, apply & bind
- Tooling (bundling)

The important stuff

- Hoisting
- Coercion
- Scope
- Context
- The Prototype
- The Event Loop
- Functions are first class
- Being Asynchronous

Developer Tools

I'll do a quick run through

Text Editor or IDE

Let's set up our editor and perhaps a linter

Tooling

- Node
- Linting and Formatting
    - ESLint
    - Prettier

The Web

I'll run through a quick intro to the structure of a page

Introduction to JavaScript

Approaching JavaScript

- JavaScript is not Java
- It can be as loose or strict as you want it
    - not strictly object-oriented
- Easy to learn but hard to master

JavaScript Traits

- Single-threaded
- Environment manages the memory for you
- Dynamically typed (with weak typing)
- Interpreted
- Prototype-based inheritance (vs. class-based)
- No built-in file access; limited I/O; safe sandbox in the web
- Weird but fun 🤪

ECMAScript

- ES3 - 1999
- ES5 - 2009
- ES6 (ES2015)
- ES2016 (ES7)
- ES2017 (ES8)
- ES2018 (ES9)
- ...
- ES.Next

Not just for the browser

- Node is a runtime for JavaScript from the command line
- Libraries (like Node) help make JavaScript a more general purpose language
- File I/O, etc
- Future version of JavaScript have proposals for memory management, etc

Why JavaScript

- It's the language of the web
- ... and server
- ... and desktop

Syntax Basics

- "C" family of languages
- Whitespace doesn't matter
- Blocks are wrapped in curly braces `{ }`
- Statements *should* be terminated by a semicolon

```
let x = 10;

if (x < 5) {
  x = 5 + 10;
}
```

Debugging in the console

- Browser's console is a line interpreter (REPL)
- All browsers have converging on the same API
- `console` object is an interface to the browser's "console"
- `debugger` triggers a breakpoint
- Can view variable scope and state

```
console.log("hi world!");
console.warn("Something bad happened");
console.table(arrayOfData);

debugger; // trigger a breakpoint
```

Exercise: Try it out

- Open your browser's developer tools
- Log something to the console

```
console.log("Hello World");

let name = "Robot Cat";

console.log(`Hello from ${name}`);
```

Values & Operators

Primitives

```
"Hello World"; // Strings
42; // Numbers
true && false; // Boolean
null; // No value
undefined; // Not yet defined
Symbol.iterator; // Symbols -- relevant more once we get into objects
```

Variables

- Variables *reference* values
- All primitives are *immutable*

```
let x = 10;
let someValue = "Hello";

someValue = 100;
```

Declaring Variables

- You can declare a set of variables in a series
- Default value is `undefined`
- Typically `camelCase`

```
let x = 10;
let a,
  b,
  c = 100;

const y = 5;

console.log(a); // undefined
```

Var, Let and Const

- `var` & `let` allow re-assignment, `const` does not
- variables are not "typed" - they can reference any value
- a `var` can be redeclared, `let` and `const` can't

```
let x = 10;
x = 5;

const element = 1;

let x = 12; // Error!
element = 2; // Error!
```

Also... *for later*

- determines the `scope` & `hoisting` behavior of the variable

Objects

- Objects are structured data
- *Properties* map to *values*
- A value can be anything
- Arrays, Functions, and pretty much everything else is type of Object

```
let user = {
  id: 5,
  username: "morris",
};

console.log(user.id); // 5
```

Functions

- Runnable blocks of code
- Have properties, such as `name` and `length`

```
// statement
function add(x, y) {
  return x + y;
}

console.log(add.length); // 2
console.log(add.name); // "add"

// expression
const add = function (x, y) {
  return x + y;
};
```

Variable Scope

- Determines what variables you can "see/access" from your current location
- Lexical (*not Dynamic*)
- Global, Function and Block scope

Global Scope

- Any variable declared outside of a function

```
var x = 10;

y = 12; // not explicitly declared
```

Function Scope

- Scope *Original*
- `var` declares a variable in `Function Scope` :

```
var x = 10;

function hello() {
  var y = 20;

  return x + y;
}

x = hello(); // x is now 30
```

Question: What are the scopes of `x`, `y` and `result` here?

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

function hello(someValue) {
  var y = 20;

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

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

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

Block scope

- ES6 introduced block-scope with `let` and `const`
- Scoped to any `{}` block
    - Objects are *Not* blocks

```
let x = 10;

if (x < 0) {
  let y = 11;
}

function hello() {
  let z = 12;
}
```

Question: Will this play nice?

```
let x = 10;
if (x < 100) {
  let y = 20;
}

function add(z) {
  return x + y + z;
}

add(5); // 35...?
```

Hoisting

- Not all variables are created equally
- `var` will be *hoisted* to the top of function *blocks*
- `let` and `const` are *not* hoisted

Question: What will be logged to the console?

```
function init() {
  x = 10; // I set the value

  var x; // THEN I declare it..?

  console.log(x); // what will I log?
}
```

Question: And this time?

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

  var x = 10; // I declared & set value after using it...?
}
```

Function Hoisting

- `function` *statements* are hoisted, too

```
statement(); // this is valid
expression(); // this produces an error

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

Arrays

- Serialized data
- Indexed from `0`
- Have methods & properties, like `length`
- Mutable content… like all objects

```
let data = [55, 12, 32];

data[0]; // 55

data.length; // 3
data.pop(); // 32

console.log(data); // [55, 12]
```

Numbers

- `Number` object
- 64 bit floating point
  - You lose precision with decimals & large numbers
  - this is not specific to JS
- Special numbers: `NaN` & `Infinity`

```
console.log(5 + 5.5); // 10.5
console.log(0.1 + 0.3); // 0.30000000000000004

typeof NaN; // "number"
NaN == NaN; // false
1 / 0; // Infinity
```

Strings

- `"double"`, `'single'` or `` `back-ticks` `` all work
- The back-tick enables string interpoloation
  - "String Template Literal"
- Strings have methods & properties, like most things in JS

```
let firstString = "Hello";

console.log(firstString + " World");
console.log(`${firstString} World`);

firstString.length; // 5
```

Comments

- Single-line with `//`
- Multi-line with `/* */`

Operation	Operators
Arithmetic	+ - * / % **
Shortcut	+= 0= *= /= %/ **=
Inc/Dec	x++ x-- --x ++x
Bitwise	~ % | ^ >> << >>>
Comparison	> >= < <=
Equality	== != === !==
Logic	! && | |
Object	. []
String	+

Exercise: Using Primitives

1. Open the following file: `src/www/js/primitives/primitives.js`
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

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 in JavaScript

- Compares values 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
Object.is(1, "1"); // false, introduced in ES6
```

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
```

Logical and / or

```
if (5 && "hello") {
  console.log("I'm in!");
}

if (0 || false) {
  console.log("I'm in");
}
```

Logical Short Circuits

These actually return values

```
// && returns first falsy otherwise last value
console.log(5 && "hello");
console.log(12 && 0);
console.log(12 && false && 50);

// || returns first truthy value otherwise last value
console.log("a" || "b" || "c");
```

Control flow

- `if`, `else if`, `else`
- `switch` statements
- `!` for negation

```
if (x) {
  // do something
} else if (!y) {
  // do something
} else {
  // do something
}
```

Switch statements

- *Should* always `break` and include a `default`

```
switch (x) {
  case 10:
    console.log("Case 10");
    break;
  case 3:
  case 2:
    console.log("Case 2 or 3");
    break;
  default:
}
```

Ternaries

```
return y < 200 ? "Value Low" : "Value High";
```

Incidentally this is often used to check if a variable is initialized

```
let x = typeof x === "undefined" ? 10 : x;
let y = y ? y : 0;

// this won't always work as expected
if (x === undefined) {
  // if x is undeclared this will error
}
```

Iterating

- `for`
- `for...in` for object properties
- `for...of` for all `Iterables`
- `while`, `do...while`
- `break`, `continue`

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

// simple for loop
for (let i = 0; i < data.length; i++) {
  console.log(data[i]);
}
```

Loops continued

```
// for..of for arrays/iterables
for (let value of data) {
  console.log(value);
}

const user = {
  id: 1,
  name: "Ryan",
};

// for..in to iterate over object properties
for (let propName in user) {
  console.log(user[propName]);
}
```

`Array` also has a built-in way to iterate, we'll see it later...

Question: What is the *scope* of `i`

```
for (var i = 0; i < data.length; i++) {
  console.log(i);
}

console.log(i); // ?
```

Exercise - Control Flow

1. Open the following file: `src/www/js/control/control.js`
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Objects

Most things in JavaScript are an object, or can behave like one

```
let x = 55;

x.toString(); // "55"

(12).toString(); // "12"
```

Dynamic properties

- Access through `.` or `[]` accessors
- You can add/remove properties at any time*

```
const box = {
  color: "red",
  height: 12,
};

box.width = 100;
box["color"] = "blue";

delete box.width;
```

Functions as Properties

- Can store any value on an object, including a `function`
- Functions are aware of the object they operate on through `this`
- Referred to as `context`

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

Abbreviated Property Definition

ES6+ introduced short-cuts to defining properties in an object (and class)

```
const bark = function () {};
const name = "Fido";

const dog = {
  id: 10,
  name,
  bark,
  sit() {
    console.log("I am sitting");
  },
};
```

Object references & mutability

- *Objects* are mutable and are passed by reference
- `===` is true only when the object is the same instance

Question: What happens to `box` here?

```
const box = { sides: 4 };

function mutator(obj) {
  obj.mutated = true;
}

mutator(box);

console.log(box); // ?
```

Question: So is this true or false?

```
const box = { sides: 4 };

function mutator(object) {
  console.log(object === box); // ?
}

mutator(box);
```

Object Property Descriptors

- `Object.defineProperty` to configure additional property behaviors

```
let obj = {};
Object.defineProperty(obj, "someName", {
  value: 42,
  configurable: false,
  enumerable: false,
  writable: false,
  //get: function() {},
  //set: function(val) {},
});
```

Object Reflection

- `typeof {}`
- Iterate with `for...in`
  - Warning: unspecified order of properties

```
for (let propName in cat) {
  console.log(cat[propName]);
}
```

Object Ownership

- Objects can have *own* properties or *inherited*
- Check property ownership with `obj.hasOwnProperty(propName)`

```
let cat = {
  legs: 4,
};

cat.toString(); // "[object Object]"

// so... does it have it?
cat.hasOwnProperty("toString"); // ?
```

Exercise - Copy objects

1. Open the following file: `src/www/js/copy/copy.js`
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Hints:

- `for (let prop in someobj) { /* ... */ }`
- `someobj.hasOwnProperty(prop)`

Cloning and merging objects

- `Object.assign`
- The spread `...` operator

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

// clone it!
const copiedObject = Object.assign({}, originalObject);

// or another way...
const anotherCopy = {
  ...originalObject,
};
```

Object Methods

- Get information out of your object
- Own, Enumerable, etc...

```
Object.keys(obj);
Object.values(obj);
Object.entries(obj);

// an array of properties that the object "owns"
Object.getOwnPropertyNames(obj);
```

Built-in Objects

- String
- Number
- Math
- Date
- Array
- RegExp

Numbers

Constants:

- `Number.MAX_VALUE`
- `Number.NaN`
- etc...

Generics:

- `Number.isInteger(n);`
- `Number.parseInt(n);`
- etc...

Instance methods:

- `num.toString();`
- `num.toFixed();`
- etc...

Strings

- `str.length`
- `str.charAt(i)`
- `str.concat()`
- `str.indexOf(needle)`
- `str.slice(iStart, iEnd)`
- `str.substr(iStart, length)`
- `str.replace(regex|substr, newSubStr|function)`
- `str.toLowerCase()`
- `str.trim()`

Math object

Constants:

- `Math.E`
- `Math.PI`
- etc...

Generics:

- `Math.abs(n)`
- `Math.pow(n, e)`
- etc..

The Date Object

- Represent a point in time
  - Must be *constructed*
- Months start at `0`, days start at `1`

```
let d = new Date(); // today
d = new Date("Wed, 20 Jan 2020 13:30:00 EST");

d = Date.now();
d = Date.UTC();

d.getTime(); // unix timestamp
d.getMonth();
d.getHours();

d.setYear(1990);
```

Arrays

Sequential data, order is maintained

Instance methods

- `arr.shift`, `unshift`, `push`, `pop`
- `concat`, `slice`, `splice`
- `indexOf`, `find`
- `sort`, `reverse`
- `every`, `some`
- `map`, `filter`, `reduce`

Generics

- `Array.isArray(a)`
- etc...

Functions

Functions in JavaScript

- Three ways to author a function:
    i. Statement
    ii. Expression (anonymous)
    iii. Arrow (static context)

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

typeof statement; // function

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

const arrow = () => a + b;
```

Function Arguments

- All arguments are available in `arguments` property
- Missing values will be `undefined`
- No function overloading in JS

```javascript
function logAll(a) {
  console.log(arguments);

  // pre-es6 days... now you'll see "rest" in use
  const args = Array.prototype.slice.call(arguments);
}

logAll(1, 3, "hi");
```

Function Defaults (*finally*)

```
function tryDefaults(a, b, c) {
  a = typeof a === "undefined" ? 1 : a;
  b = typeof b === "undefined" ? 10 : b;
  c = typeof c === "undefined" ? a + b : c;
  return a + b + c;
}
```

becomes…

```
function tryDefaults(a = 1, b = 10, c = a + b) {
  return a + b + c;
}

tryDefaults(undefined, 12);
```

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...
});
```

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) {
  // 1. `acc' is the accumulator
  // 2. `elm' is the current element
  // 3. You must return a new accumulator return acc + elm;
}, 0);

sum; // 6
```

Functional JS

In this way we can break down our code into reusable, testable function components.

```
const names = ["abe", "bob", "carol"];
let allNames = "";
for (let i = 0; i < names.length; i++) {
  allNames += ` ${names[i]}`;
}
```

```
const nameReducer = (acc, name) => {
  return `${acc} ${name}`;
};
let allNames = names.reduce(nameReducer);
```

Exercise: Arrays & Functional Programming

1. Open the following file:

   `src/www/js/array/array.js`

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

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

Function Patterns

Anonymous Functions

```
let anon = function() {};
```

- A function expression without a name
- Difficult to test in isolation
- Discourages re-use
- Can still be given a name but it's not available outside the function

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

Functions as Callbacks

- When a function is provided as an argument as something to be invoked inline, or under specific circumstances (like an event)

```
function runCallback(callback) {
  // does things
  return callback();
}
```

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

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);
```

Closures

- Extremely common in JavaScript
- When the *outer scope* of a function *closes over* the *inner scope*

```javascript
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
```

Closures for Privacy & State

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

  return {
    getPrivateVar: function () {
      return privateVar;
    },
    setPrivateVar: function (n) {
      if (n % 2 === 0) {
        privateVar = n;
      }
    },
  };
};

let x = Foo();
x.getPrivateVar(); // 42
```

Question: How might you avoid initializing this closure?

```
const Foo = function () {
  // ...
};

let x = Foo(); // <--- can we change this or avoid it?
x.getPrivateVar();
```

The IIFE

Immediately Invoked Function Expression

```
const x = (function () {
  // ...
})();
```

Commonly seen for:

- Initalizing an old-world module
- Initializing a stateful singleton
- Protecting / Clean Scope

Exercise: Sharing Scope

1. Open the following file:

   `src/www/js/closure/closure.js`

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Closure Gotcha

Question: What will this output and in what order?

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

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

Scope & Context

Adding Context to a Scope

- We already discussed **scope**
  - Visibility of variables
  - Lexical (you can read it and determine scope)
- There is also **context**
  - Based on location a function was invoked
  - Dynamic, determined at runtime
  - Accessible via `this`

Calling functions through objects

```
let apple = {
  name: "Apple",
  color: "red",
};
let orange = {
  name: "Orange",
  color: "orange",
};

let logColor = function () {
  console.log(this.color); // <!-- _this_ is the context...
};

apple.logColor = logColor;
orange.logColor = logColor;

apple.logColor(); // "red"
orange.logColor(); // "orange"
```

Context and `this`

- `this` is a keyword
- References the "object of invocation"
- Allows a method to reference an object instance
- Single methods can service many objects
- Central to prototypical inheritance in JS

Control the scope

- Bound at runtime when a function is invoked
- Can be set manually with `call`, `apply` and `bind`
- But, Arrow functions are different...
  - static, use their parent function's context

Question: What is the difference in behavior between `hello` and `world`?

```
const hello = function() {
  console.log(`${this.name} is ${this.color}`)
};

const world = () => {
  console.log(`${this.name} is ${this.color}`);
}
let orange = {
  name: "Orange",
  color: "orange",
  hello,
  world
}
```

OO in JS (Intro only)

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.

```javascript
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: `src/www/js/constructors/constructors.js`
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

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 on class

- Getters/Setters
- Statics
- Super() calls

Exercise: Class Upgrade

1. Re-open the following file: `src/www/js/constructors/constructors.js`
2. Convert your Constructor Function to use the Class keyword instead
3. All tests should continue to pass

Errors in JS

Exception Basics

- Errors propagate as exceptions
- try, catch, throw and finally
- only catch synchronous, run-time errors

Throwing Exceptions

```
if (somethingGoesWrong) {
  throw "This went wrong";
}

if (somethingElseGoresWrong) {
  throw new Error("Something ent wrong");
}
```

Catching Errors

```
try {
  // try something...
  return;
} catch (e) {
  if (e instanceof MyCustomError) {
    throw e; // you can re-throw
  }
} finally {
  // runs even if the try/catch returns!
  // clean up
}
```

Built-in errors

- Error - generic
- ReferenceError - variable use
- SyntaxError - error parsing
- TypeError - variable not expected type
- etc...

Custom Errors

Just extend the error class

```
class MyCustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "MyCustomError";
  }
}
```

Exercise: Exceptions

optional

1. Open the following file: `src/www/js/exceptions/exceptions.js`
2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

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>
```

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

```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.parentNode;
```

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>
```

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

DataSet API

```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:

```
src/www/js/flags/flags.js
src/www/js/flags/index.html (read only!)
```

2. Open the `index.html` file in your web browser.

3. Complete the exercise.

Events

The Event Loop

- Single-threaded, asynchronous event model
- 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: /js/runtime/

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
  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:

```
src/www/js/events/events.js
src/www/js/events/index.html (read only!)
```

2. Open the index.html file 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
```

Exercise: Making Ajax Requests with XHR

1. Open the following files:

```
src/www/js/artists/artists.js
src/www/js/artists/index.html (read only!)
```

2. Open http://localhost:3000/js/artists/

3. Complete the exercise, using our internal API:

   GET http://localhost:3000/api/artists

   GET http://localhost:3000/api/artists/1

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 the data or error respectively

```
const delayed = function () {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      if (true) {
        resolve(100);
      } else {
        reject(0);
      }
    }, 500);
  });
};
```

then there is the promise consumer...

Promise Consumer

- `then()`, `catch()`, `finally()` (soon)
- You can chain these
- You can keep using the promise

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

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

someOtherThingThatWorksWithPromises(delayed);
```

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);
  });
```

Exercise: Using the Fetch API

1. Start your server if it isn't running
2. Open `src/www/js/fetch/fetch.js`
3. Fill in the missing pieces
4. To test and debug, open
   localhost:3000/js/fetch/

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)

Testing JavaScript

- We'll use 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 `src/www/js/jasmine/adder.spec.js`
2. Read the code then do exercise 1 (we'll do exercise 2 later)
3. To test and debug, open `src/www/js/jasmine/index.html`

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 `src/www/js/jasmine/adder.spec.js`
2. Read the code then do exercise 2
3. To test and debug, open `src/www/js/jasmine/index.html`

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 `src/www/js/jasmine/delayed.spec.js`
2. Read the code then do exercise 3
3. To test and debug, open `src/www/js/jasmine/index.html`

Resources

Get more

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