Advanced JavaScript

Ryan Morris
Develop Intelligence

Introductions

- Who am I?
- Who are you?
  - What do you do?
  - What do you hope to gain from this course?
  - What is your JS/Programming Background
- What is your current development environment and process?

How the class works

- Lecture & Labs
- Informal
  - Stop me anytime for questions or more information
  - Daily outline is flexible
    - **There is too much to cover** so we'll adjust as needed
- Class review on day 2

Get the most out of the class

- Ask questions!
- Do the labs (pair up if needed)
- Be punctual
- Avoid meetings and work distractions
- Master your google-fu
- Play along in the console
- Don't be afraid to break stuff

Goals for this class

- Become ***more*** familiar with:
  - Functions in JavaScript
  - Object-Oriented JavaScript
  - Modules
  - Asynchronous Programming
- Learn about:
  - Javascript workflows
  - Using libraries such as jQuery, Underscore, and more
  - Popular MVC frameworks
- Be able to:
  - Build a simple single-page MVC application

Resources

- Documentation
  - http://devdocs.io
  - https://developer.mozilla.org/en-US/docs/Web
  - http://kapeli.com/dash (Mac only)
  - Google it.
- Compatibility checks
  - http://caniuse.com
- ES 5 compatibility table
  - http://kangax.github.io/compat-table/es5/

Tools

- IDE/Editor
  - WebStorm http://www.jetbrains.com/webstorm
  - SublimeText http://www.sublimetext.com
  - Notepad++ http://notepad-plus-plus.org
- Lab Package
  - https://github.com/rm-training/advanced-js
  - Grab it and unpack it
- Node.js
  - npm
- Console in the browser

👆 Everyone OK with the above?

PDF for today

- In case you didn't quite catch something, or can't see the screen well:
  - ##TODO - LINK TO PDF##

Agenda

- Day 1
  - Language recap
  - Functions in JavaScript
  - Object-Oriented JavaScript
  - How to create modules
  - Asynchronous callbacks and promises
- Day 2
  - Class Lab
  - ES6
  - Web APIs
  - Frameworks
  - Lay of the land

Pop Quiz!

- Let's review core concepts
- Use your console to test along with me

Do we need to review the console 💻?

Exercise: Hoisting (pt 1 of 3)

- What will the output be?

```
function foo(x) {
 x = 42;
 var x;

 console.log(x); // ?
 return x;
}
```

Exercise: Hoisting (pt 1 of 3)

This…

```
function foo(x) {
 x = 42;
 var x;

 console.log(x);
 return x;
}
```

Becomes…

```
function foo(x) {
 var x;
 x = 42;

 console.log(x);
 return x;
}
```

Exercise: Hoisting (pt 2 of 3)

- And this?

```
function foo(x) {
 console.log(x); // ?
 var x = 42;
 return x;
}
```

Exercise: Hoisting (pt 2 of 3)

This…

```
function foo(x) {
  console.log(x);
  var x = 42;
  return x;
}
```

Becomes…

```
function foo(x) {
  var x;
  console.log(x);
  x = 42;
  return x;
}
```

Exercise: Hoisting (pt 3 of 3)

- And finally

```
foo(); // ?
bar(); // ?

function foo() {
 console.log(“Foo!”);
}

var bar = function(){
 console.log(“Bar!”);
}
```

Exercise: Hoisting (pt 3 of 3)

This…

```
foo();
bar();

function foo() {
 console.log(“Foo!”);
}

var bar = function(){
 console.log(“Bar!”);
}
```

Becomes…

```
var x;
function foo() {
 console.log(“Foo!”);
}

foo();
bar();

bar = function(){
 console.log(“Bar!”);
}
```

Exercise: Variable scope

- What is the scope of **w**, **x**, **y** and **z**?

```
function foo(x) {
    var y = 0;
    if (x === 1) {
        var z = 1;
        w = x;
    }
}
```

Exercise: Callbacks & Async

- What does this code do?

```
for (var i = 1; i <= 5; i++) {
   setTimeout(function() {
      console.log(i);
   }, i * 1000);
}

// output will be?
```

Exercise: Objects

- What is going on here?

```
var x = {
 color: “magenta”
}
x.name = “Bob”;
var y = {};

for (var prop in x) {
  if (x.hasOwnProperty(prop)) {
    y[prop] = x[prop];
  }
}
```

Exercise: Mutability

- What will the result be to x?

```
var color=“Red”,
  x = {color: “magenta”};

function setColor(obj, color) {
  obj.color = color;
  color = color;
}

setName(x, “Blue”);
console.log(x, y);
```

Exercise: Functions and Context

- What is going on here?

```
var x = {color: “magenta”}
var y = {color: “orange”}

var z = function() {
 console.log(“My color is”, this.color);
}

x.log = y.log = z;

x.log(); // ?
y.log(); // ?
z(); // ?… for bonus points
```

Quick Refresher

- There are 5 primitive types (string, number, boolean, null, undefined) and then Objects
  - Functions are a callable Object
  - Objects are property names referencing data
  - Arrays are for sequential data
- Declare variables with “var”
  - **Block scope**
- Types are coerced
  - Including when a primitive is used like an object
- *Almost Everything* is an object, except the primitives
  - despite them having object counterparts

Key Concepts

- Hoisting
- Function Scope (it’s lexical)
- Context, *this* keyword (it’s dynamic)
- Functions are first-class
- Call-by-sharing
    - Objects are mutable

module

THE DOM

The DOM Refresher

- How does everyone feel about
  - HTML syntax
  - CSS selector syntax
  - DOM methods

Document Object Model

- Browser parses HTML and builds a model of the structure, then uses the model to draw it on the screen
- "Live" data structure
- What most people hate when they say they hate JavaScript
- The browser's API it exposes to JavaScript for interfacing with the document

DOM Structure

- Global **`document`** variable gives us programmatic access to the DOM
- It's a tree-like structure
- Each node represents an **element** in the page, or **attribute**, or **content** of an element
- Relationships between nodes allow traversal
- Each DOM node has a **`nodeType`** property, which contains a code for the type of element…
  - 1 – regular element
  - 3 – text

Accessing individual elements

- Starting at **`document`** or a previously selected element
- **`.getElementById`**`(“main");`
  ```
  // returns first element with given id
  // <div id=“main”>Hi</div>
  ```
- **`.querySelector`**`("p span");`
  ```
  // returns first matching css selector
  // <p><span>Me!</span><span>Not!</span></p>
  ```

Accessing element lists

- Using **`document`** or a previously selected element
- These return a `NodeList`
- **`.getElementsByTagName`**`("a");`
  ```
  // all <a> elements
  ```
- **`.getElementsByClassName`**`("fancy");`
  ```
  // all elements with specified class
  // <span class=“fancy”></span>
  ```
- **`.querySelectorAll`**`("p span");`
  ```
  // all elements that match the css selector
  // <p><span>Me!</span><span>Me!</span></p>
  ```

Traversal

- Move between nodes via their relationships
- Element node relationship properties
  - `.parentNode`
  - `.previousSibling, .nextSibling`
  - `.firstChild, .lastChild`
  - `.childNodes // NodeList`
- But… mind the whitespace!

Node Types

- Nodes can be of different types, we are mostly concerned with element nodes…

  -
    ```
    anElement.nodeType
    // 1 = Element
    // 3 = Text node
    // 8 = Comment node
    // 9 = Document node
    ```

Element Traversal

- Avoid’s text-node issues
- Supported in ie9+
- From an element node
  - `.children`
  - `.firstElementChild, .lastElementChild`
  - `.childElementCount`
  - `.previousElementSibling`
  - `.nextElementSibling`

Collections

- **`HTMLCollectionObject/NodeList`**
  - An array-like object containing a collection of DOM elements
  - The query is re-run each time the object is accessed, including the **length** property

Creating new nodes

- **document.createElement**(“div")
  - creates and returns a new node without inserting it into the DOM
- **document.createTextNode**("foo bar”)
  - creates and returns a new text node with given content

Adding nodes to the tree

```
// given this set up
var parent = document.getElementById(“users”),
    existingChild = parent.firstElementChild,
    newChild = document.createElement(“li”);

document.appendChild(newChild);
// appends child to the end of parent.childNodes

document.insertBefore(newChild, existingChild);
// inserts newChild in parent.childNodes
// just before the existing child node existingChild

document.replaceChild(newChild, existingChild);
// removes existingChild from parent.childNodes
// and inserts newChild in its place

parent.removeChild(existingChild);
// removes existingChild from parent.childNodes
```

Element Attributes

- Accessor methods
  - `.getAttribute("title");`
  - `el.setAttribute("title", "Hat");`
  - `el.hasAttribute("title");`
  - `el.removeAttribute("title");`
- As properties
  - `.href`
  - `.className`
  - `.id`
  - `.checked`

Element content

-
  ```
  el.textContent;
  // text content of node and all children
  ```

-
  ```
  el.innerHTML;
  // html content of node and all children
  ```

-
  ```
  el.nodeValue;
  // text, comment, attribute node values
  ```

-
  ```
  el.value;
  // form input values
  ```

Events

- Single-threaded, asynchronous event model
- Events fire and trigger registered handler functions
- Events can be click, page ready, focus, submit, etc

Event Handling

- Use the **addEventListener** method to register a function to be called when an event is triggered

```
var el = document.getElementById("main");

el.addEventListener("click", function(event) {
     console.log(
          "event triggered on:",
          event.target
     );
}, false);

// not onClick properties
```

Event handler context

- Functions are called in the context of the DOM element

```
el.addEventListener(“click”, myHandler);

function myHandler(event) {
  this; // equivalent to el
  event.target; // what triggered the event
  event.currentTarget; // where listener is bound
}
```

Event Propagation

- An event triggered on an element is also triggered on all “ancestor” elements
- Two models
  - Trickling, aka Capturing (Netscape)
  - Bubbling (MS)
- Event handlers can affect propagation

```
// no further propagation
event.stopPropagation();

// no browser default behavior
event.preventDefault();

// no further handlers
event.stopImmediatePropagation();
```

Event Delegation

- Where an event handler on a parent element handles all events that *bubble up* from it’s child elements

```
document
  .querySelector(‘ul’)
  .addEventListener(“click”, myLiHandler);

function myLiHandler(event) {
  if (e.target && e.target.nodeName == "LI") {
    console.log(
      e.target.innerHTML, " was clicked!”
    );
  }
}
```

Complete event handling example

```
var el = document.getElementById(‘some-id’);

el.addEventListener(‘click’, function(event) {
        // “this” represents the element handling the event
        this.style.color: “#ff9900”;

        // “target” represents the element that triggered
        event.target.style.color: “#ff9900”;

        // you can stop default browser behavior
        event.preventDefault();

        // or you can stop the event from bubbling
        event.stopPropagation();

});
```

Exercise - DOM Warmup

1. Start the node JS Server

```
node bin/server.js
```

2. Open the following URL:

```
http://localhost:3000/warmup/
```

3. Open the following JavaScript files:

```
www/warmup/index.html
www/warmup/warmup.js
```

4. Follow the instructions
5. Hint: Use MDN as an API reference

module

FUNCTIONS

Functions in JavaScript

- JavaScript is multi-paradigm, including functional
- Functions are first-class objects in JavaScript
  - Instances of the **Function** object
  - Have state and methods
  - Can be referenced by variables
  - And passed into other functions (higher-order!)

Function Usage

- Being first-class objects, they support
  - Anonymous/Lambda
  - Closures
  - IIFEs
  - Context Binding and Chaining
  - Partial Application

Functions as first-class objects

- So you can do crazy things like this:

```
var add = function(x, y) {
    return x + y;
}

// assign it around
var adder = add;

console.log(add(1,2));
console.log(adder(1,2));
```

Functions as first-class objects

- And:

```
function add = function(x,y,z) {}

var fnCaller = function(fn, …args) {
  fn(…args);
}

// pass it around
console.log(fnCaller(add, 1, 2, 3));
```

Anonymous Functions

- A function defined via **expression** and assigned to a variable

```
var x = function () {}
```

- The function can be passed around
- One of the most useful and powerful features of JavaScript
- You should still ***label** it*

```
var x = function myLabel() {}
```

Anonymous Functions

```
var add = function(x, y, cb) {
  cb(x + y);
};

add(10, 20, function(sum) {
  console.log(sum); // 30
});

// label an anonymous function
var add = function add(x, y) {}

$element.on(‘click’, function handleElClick (e) {}
```

Function **`arguments`**

- Functions have access to a special internal when invoked, **`arguments`**
  - contains all parameters passed to the function
  - an *array-like* object
    - needs to be converted to an array to get all the array-methods

Function arguments

```
function doSomething() {
  // call an array method with
  // with arguments as the function context
  var args = Array.prototype.slice.call(arguments);

  // or in ES6
  var args = Array.from(arguments);

  console.log(args);
}

doSomething(1, 2, 3); // ?
```

Function **`this`**

- Functions also have access to an additional internal variable upon invokation, **`this`**
- Refers to the context of the function at call-time
  - Dynamically bound (not lexical)
  - “The object in context that invoked the method”
- Necessary for
  - Inheritance
  - Multi-purpose functions
  - Method awareness of their objects

this example

```
var person = {
  name: “John Doe”,
  speak: function() {
    console.log(“Hi my name is”, this.name);
  }
}

person.speak(); // ?
var speak = person.speak;
speak(); // ?

// and if we put it on another object?
var otherPerson = {name: “Jim”}
otherPerson.speak = person.speak;
otherPerson.speak(); // ?
```

Setting function context

- You can set the context of a function by force
- `Function.prototype.call(obj, arg1, arg2, ..)`
- `Function.prototype.apply(obj, [arg1, arg2, ..])`

Setting function context

```
var speak = person.speak;

// invoke speak in the context of person
speak.call(person);
speak.apply(person);

// invoke speak in the context of otherPerson
person.speak.call(otherPerson);
```

Binding function context

- You can hard-code* a function’s context
  - `Function.prototype.bind(obj, arg1, arg2, ..)`
- Just creates a copy with a hard-coded context

Binding function context

```
// permanently bound to person object
var speak = person.speak.bind(person);
speak();

// and if we put it on another object?
var otherPerson = {name: “Jim”};

otherPerson.jimSpeak = person.speak.bind(person);
otherPerson.jimSpeak(); // ?
```

A practical example of bind()

```
var person = {
  name: “Human”,
  speak: function() {
    console.log(“Hello from “, this.name);
  }
}

var button = document.getElementById(‘myButton’);
// callback won’t be called in the object’s context
button.addEventListener(
  ‘click’,
  person.speak
);

// instead we can:
// person.speak.bind(person)
// function() {person.speak()}
// or closures…
```

Function Partials

- Create a new function from an existing one, with one or more of its arguments already defined:

```
function add(x, y) {
      return x + y;
}
add(1, 2);  // 3

// create a new function that has bound arguments
// notice, there is no context being bound…
var add10 = add.bind(null, 10);
add10(2); // 12
```

Exercise - Better Partials

1. Open the following JavaScript file:
   `www/partial/partial.js`
2. Get the tests to pass:
   `node bin/jasmine spec/partial.spec.js`
3. Implement a Function.prototype.partial function that gets the following code to work
4. *Bonus*: Get it to work with any # of additional arguments?

```
var obj = {
  magnitude: 10,
  add: function (x, y) {
    return (x + y) + this.magnitude;
  }
}
var add10 = obj.add.partial(obj, 1);
add10(2); // should return 30
```

module

FUNCTION PATTERNS

IIFEs

- **I**mmediately **I**nvoked **F**unction **E**xpression
- A function that is defined within a parenthesis, and immediately executed

```
(function() {
   var x = 1;
   return x;
})();
```

IIFE Uses

- Define namespaces/modules/packages
- Creates a scope for private variables/functions
- Extremely common in JS

Privacy and modules with IIFEs

```
var helper = (function() {
  var x = 1; // effectively private
  return {
    getX: function() {
      return x;
    },
    increment: function() {
      return x = x + 1;
    }
  }
})();

helper.getX();
helper.increment();
```

Privacy and modules with IIFEs

```
var helper = (function($) {
  var $el = $(‘button’);
  return {
    getElement: function() {
      return $el;
    },
    clearElement: function() {
      $el.html(‘’);
    }
  }
})(jQuery); // pass in globals
```

Exercise - Using IIFEs to make private functions

1. Open the following JavaScript file:

   ```
   www/hosts/hosts.js
   ```

2. Follow the instructions inside the file
3. Get the tests to pass:

   ```
   node bin/jasmine spec/hosts.spec.js
   ```

Closures

- A **closure** is created when an inner function has access to an outer (enclosing) function's variables
- A function that maintains state (it’s outer scope) after returning
- It has access three scopes:
  - Own – variables defined in its body
  - Outer – parameters and variables in the outer function
  - Global
- Pragmatically, *every* function in JavaScript is a closure!

Closure Example

```
function closeOverMe() {
    var a=1; // effectively private
    return function iCloseOverYou() {
        console.log(a);
    };
};
var witness = closeOverMe();
witness(); // 1
```

Closure Module Example

```
var helper = (function() {
  var secret = “I am special”;

  return {
    secret: secret,
    tellYourSecret: function() {
      console.log(secret);
    }
  }
})();

helper.tellYourSecret(); // ?
helper.secret = “New secret”;
helper.tellYourSecret(); // ?
```

Function Chaining

- Fluent style of writing a series of function calls on the same object
  - By returning context (**`this`**)

```
"this_is_a_long_string"
     .substr(8)
     .replace('_', ' ')
     .toUpperCase(); // A LONG STRING
```

Support function chaining

```
var Cat = {
    color: null,
    hair: null,
    setColor: function(color) {
        this.color = color;
        return this;
    },
    setHair: function(hair) {
        this.hair = hair;
        return this;
    }
};

Cat.setColor('grey').setHair('short');
```

Exercise: What’s wrong here?

```
// function that returns a month name
// given an integer representing the month
var monthName = function(n) {

  var names = [“jan”, “feb”, “mar”, /*all the
months */];

  return names[n] || “”;

}
```

Lazy Function Definition

```
var monthName = function(n) {

  var names = [“jan”, “feb”, “mar”];

  // we are re-assigning the var to a new fn!
  // the new function will behave as a closure
  var monthName = function(n) {
    return names[n] || “”;
  }

  return monthName(n);

}
```

Functions Recap

- Are **Objects** with their own methods and properties
- Can be **anonymous**
- Can be bound to a particular **context**, or particular **arguments**
- Can be **chained** together, provided the return of each function has methods
- **Closures** can be used to maintain access to calling context's variables
- **IIFEs** can be used to maintain internal state
  - Both closures and IIFEs can be used to simulate "private" or hidden variables

module

OBJECT ORIENTED JAVASCRIPT

~~OO JS~~ - Object Creation in JavaScript

- There's no "one" way in JavaScript
  - A rabbit hole of approaches
  - 4 competing JS engines, a lot of compromise in the definition of the language
- Lot's of people trying to emulate classical styles
  - Your soul *may* want JS to be like other OO-approaches
- Resist the urge to say, "where's my classes"…
  - Accept that there is "no right way"…
  - Learn about the many ways to create objects…
  - *Then decide which way to go with your team*

Object Creation in JavaScript

- Object literal
- **`Object.create()`**
- Constructors w/ **`new`**
- Factory Functions
- ES6 **`class`** keyword

Let’s begin the OO Journey

- We create objects that represent the *things* of our system
  - They have methods for behavior
  - And properties for data
  - …

The Object Literal

```
// We create Objects to represent Things in our
// system, each with methods and properties

var dog = {
  talk: function() {
    console.log(“Bark!”);
  }
}

var cat = {
 hasAttitude: true,
 talk: function() {
    console.log(“Meow!”);
  }
}
```

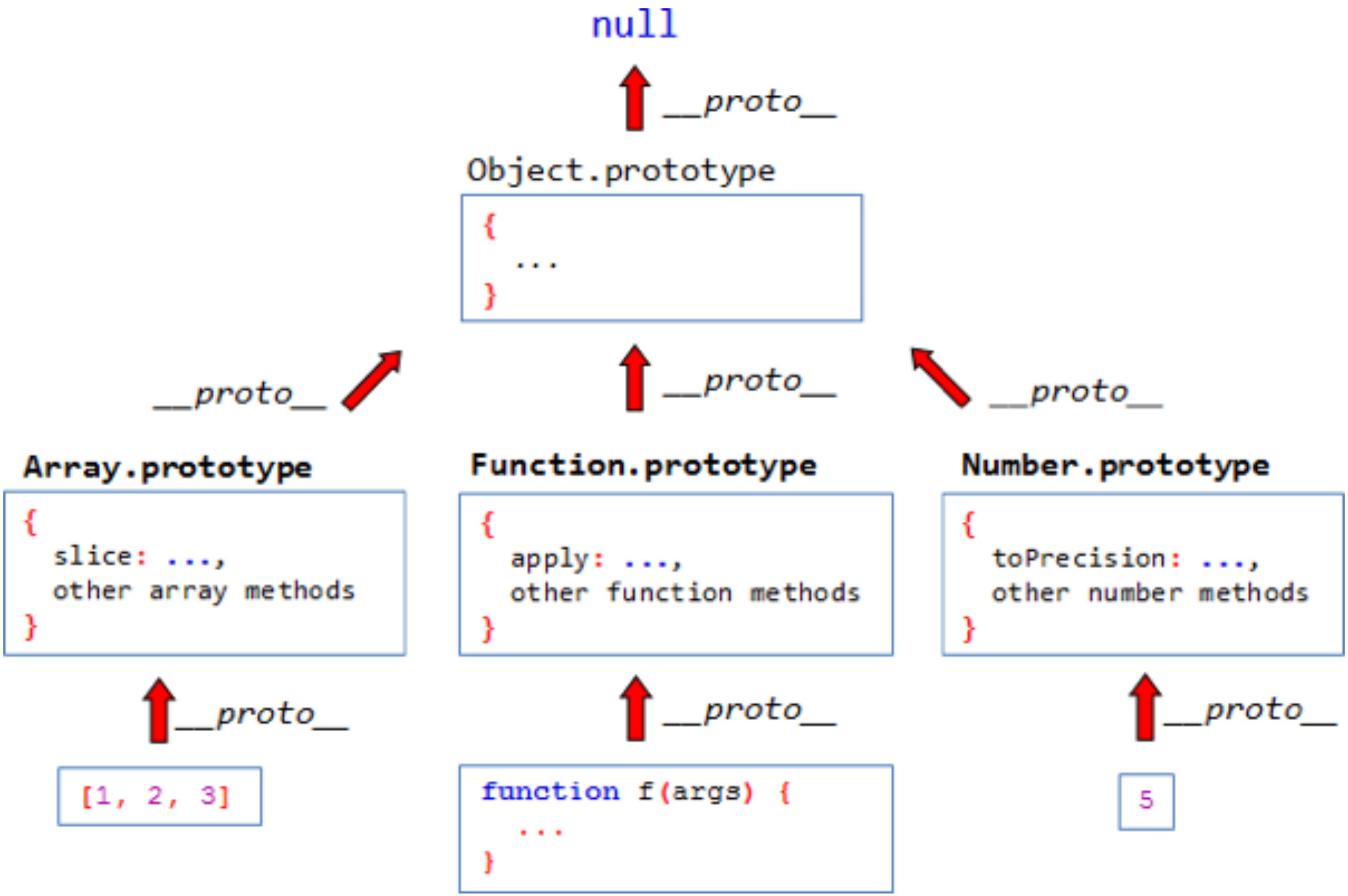
Prototypal Inheritance

```
// abstracting out shared behavior
var animal = {
  talk: function() {
    console.log(this.sound + “!”);
  }
}

// create an object with animal as it’s prototype
var dog = Object.create(animal);
dog.sound = “bark”;

var cat = Object.create(animal);
cat.hasAttitude = true;
cat.sound = “meow”;
```

Prototype

- **Prototype** – “an original or first model of something from which other forms are copied or developed”
- Objects have an internal link to another object called its *prototype*
- Each prototype has its own prototype, and so on, up the ***prototype chain***
- Objects ***delegate*** to other objects through this prototype linkage
  - “For this object, use this other object as my delegate”

Prototypes Visualized



null

__proto__

Object.prototype

```
{
  ...
}
```

__proto__

__proto__

__proto__

Array.prototype

```
{
  slice: ...,
  other array methods
}
```

Function.prototype

```
{
  apply: ...,
  other function methods
}
```

Number.prototype

```
{
  toPrecision: ...,
  other number methods
}
```

__proto__

__proto__

__proto__

```
[1, 2, 3]
```

```
function f(args) {
  ...
}
```

```
5
```

Prototype Augmentation

- The linkage is live, you can extend at run-time and affect all copies

```
var animal = {};

var dog = Object.create(animal);

// setting a property on the prototype of dog
animal.hasTail = true;

console.log(dog.hasTail); // ?
```

.prototype vs. __proto__

- **`.prototype`** is a property of the Function object
  - Every Function object has one
  - When a function is used as a constructor, new objects will point to **`.prototype`** as their “prototype”
  - “*When I create an Array instance, it delegates to Array.prototype*”
- **`.__proto__`** is an instance property of an object
  - References its “prototype”
  - Prototype Chain
  - “*When I create an Array instance, use an internal property `__proto__` to point to Array.prototype*”
  - Not standard until ES6

Prototype vs Class

- JavaScript leverages **prototypal inheritance** instead of **class-based** inheritance
- Classes…
  - Act as blueprints
  - You make copies
- Prototypes…
  - Act as delegates
  - Live representative, not a copy
- ES6 `class` keyword
  - Just a wrapper around prototype, so… ¯_(ツ)_/¯

Constructors & **new**

- A function that expects to be used with the ***new*** operator is said to be a constructor

```
var MyConstructor = function(name) {
  // set instance-level properties
  this.name = name;
}

// set delegated methods and properties…
MyConstructor.prototype.sayHello = function() {};

var instance = new MyConstructor(‘DogCat’);
```

- What it does exactly…
    1. Uses **`this`** to set own properties on a new object
    2. Set's the **`__proto__`** link from new object to the **`prototype`** of the function
    3. Returns the new object

Pseudo-Classical Inheritance

```
// We create a function to serve as our constructor
// which sets instance properties
var Animal = function (sound) {
  this.sound = sound;
}

// We use it’s prototype to define delegated props
Animal.prototype = {
  talk: function() {
    console.log(this.sound + “!”);
  }
}

var dog = new Animal(“bark”);
var cat = new Animal(“meow”);
cat.hasAttitude = true;
```

Constructors and Inheritance

- Depends on usage of **`new`** keyword, constructor functions and the prototype linkage
- Still… isn’t like classes
- Only supports single-inheritance
- Since inheritance is programmatic in JavaScript, we can create helpers to make things easier:
  - http://jsfiddle.net/jmcneese/p2ohmuw0

Pseudo-Classical Inheritance cont…

```
// superclass
var Animal = function (sound) {
  this.sound = sound;
}
Animal.prototoype = {/*… some stuff …*/}

// subclass
var Dog = function(breed) {
  // apply the superclass constructor
  Animal.call(this, “bark”);
  this.breed = breed;
}

// Dog extends Animal
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

var dog = new Animal(“bark”);
var cat = new Animal(“cat”);
cat.hasAttitude = true;
```

Factory Functions

- Functions that create and return objects
- Alternative to constructors
- Better encapsulation
- Retains context (through closures)

Factory Function Example

```
function dogMaker() {
  var sound = ‘woof’;

  return {
    talk: function() {
      console.log(sound);
    }
  }
}

var dog = dogMaker();
dog.talk();

// real-world practical bonus here
// this retains context and works!
setTimeout(dog.talk, 1000);
```

Object Composition

- When objects are *composed* by *what it does*, not *what it is*
  - Animal
    -> Cat
    -> Dog
    vs
  - Animal
    -> Animal + Meower
    -> Animal + Barker
- Alternative to multiple inheritance
- Properties from multiple objects are copied onto the target object

Mixins Example

```
function CatDog() {
  Dog.call(this);
  Cat.call(this);
}

// inherit one class
CatDog.prototype = Object.create(
  Dog.prototype
);

// mixin another
// Object.assign is ES6 object merging)
Object.assign(CatDog.prototype, Cat.prototype);
```

Functional Composition Example

```
var Animal = {legs: 4}

var meower = function (obj) {
  this.sound = “Meow”;
  this.purr = function() {}
}
var barker = function () {
  this.sound = “Bark”;
}

var cat = Meower(Animal);
var dog = Barker(Animal);
```

Introspection

- **`instanceof`** operator

```
[1, 2, 3] instance Array; // returns true
```

- **`.isPrototypeOf()`** function

```
Object.prototype.isPrototypeOf([1,2,3]); // true
String.prototype.isPrototypeOf([1,2,3]); // false
```

- **`Object.getPrototypeOf()`** function

```
Object.getPrototypeOf([1,2,3]); // Array.prototype
```

Object.freeze

- Can’t add new properties
- Can’t change values of existing properties
- Can’t delete properties
- Cant’ change property descriptors

```
Object.freeze(obj);

assert(Object.isFrozen(obj) === true);
```

Object.seal

- Properties can’t be deleted, added or configured
- Property values can still be changed

```
Object.seal(obj);

assert(Object.isSealed(obj) === true);
```

Object.preventExtensions

- Prevent any new properties from being added

```
Object.preventExtensions(obj);

assert(Object.isSealed(obj) === true);
```

Object.defineProperty

- Define (or update) a property and it’s configuration
- Some things that can be configured
  - enumerable
  - value
  - writable

```
Object.defineProperty(
   obj,
   propName,
   definition
);
```

OO – Recap

- No classes, only prototypes
  - Prototypes are full-fledged objects that new objects use to delegate behavior to
  - Everything derives from Object
- Fundamental concepts are fully supported
- Encapsulation/visibility can be implemented via closure/IIFE patterns
- Objects and their properties are runtime configurable
  - As are their mutability settings
  - Enough rope to hang yourself with, so be careful!

module

MODULES

Modules

- Organize logical units of functionality
- Easier to reason about the system and code
- Prevent namespace clutter and collisions
- Supports code reuse
- Supports decoupling

Namespacing with Objects

- You can use an object, which has a scope, but it lacks privacy

```
App = {
    name: 'MyApp',
    controllers: {
        People: ...
    },
    models: {
        Person: ...
    }
};
```

Namespacing with Functions

- Functions provide their own scope and encapsulation through closure

```
var dayName = (function () {
  var names = [‘sun’, ‘mon’, ‘tue’];
  return function(number) {
    return names[number-1];
  }
})();

console.log(dayName(1)); // sun
```

The Module Pattern

- Many takes on this pattern
- Typically uses IIFEs to create own scope and closures to hide private data
- Gotchas
  - Fragile - module can be modified
  - No dependency management
  - Non-performant at scale

Revealing Module Pattern

```
var Car = (function() {
  // private
  var speed = 0;

  var setSpeed = function(newSpeed) {
    if (newSpeed > 0 && newSpeed < 100) {
      speed = newSpeed;
    }
  };

  return {
    stop: function() {setSpeed(0);},
    go: function() {setSpeed(100);}
  }
})();
```

Actual Modules in JS

- It’s possible to write modular JS
  - Modules are organized in individual files (one per module)
  - Modules **export** their desired interface
  - Script(s) can then **import** desired interfaces
- Several module implementations in JS
  - CommonJS modules (node)
  - Asynchronous Module Definition (require.js)
  - ES6 modules

CommonJS

- Used in node.js
- Modules are organized in individual files
- Uses **`exports`** object to make things available and a **`require`** function to load modules
- Pros
  - Simple
  - `require` can be called anywhere
  - Handles dependencies
- Cons
  - Synchronous, not ideal for all environments
  - Requires compilation step to be used in web (webpack, browserify)

```
// bar.js
module.exports = function() {
  console.log(‘I am module!’);
}
```

```
// app.js
var bar = require(‘./bar.js’);
bar();
```

CommonJS Examples

```
// bar.js
var song = ‘My favorite song’;
module.exports.foo = function() {}
module.exports.bar = function() {}


// app.js
var bar = require(‘./bar.js’).bar;
var foo = require(‘./bar.js’).foo;

// or in ES6
const {foo, bar} = require(./bar.js);

// note what is available
console.log(song); // ReferenceError!
```

Asynchronous Module Definition

- Use **`define()`** function to define modules and **`require()`** to handle dependency loading
- Main difference is it support for asynchronous loading
- Has a complete picture of the dependency graph at all times
- Implemented by RequireJS and Dojo
- Pros
  - Multiple modules can be loaded in parallel or lazily
  - Perfect for web-applications
  - Dependencies can be loaded anytime, as needed
- Cons
  - Bit more complex
  - Loader libraries are required
  - Not as nicely organized (app is implemented in require/ defines)

AMD Example

```
// define modules with DI
define(‘myMod’, [‘foo’, ‘bar’], function(foo, bar) {
  // any return value is the module export
  return {
    methodOne: function() {foo.doSomething()}
  }
});

// load module(s)
// typically in the app.js space
// or inline of another module
require([‘myMod’], function(myMod) {
  myMod.methodOne();
});
```

ES6 Modules

- Similar to CommonJS
- Supports synchronous and asynchronous loading
- Name exports with **`export`** directive
- **`import`** directive bring modules into the namespace
  - imports bindings, not values
- Static module structure
  - Can determine imports/exports lexically
  - Can’t use import just anywhere or dynamically

```
// bar.js
export function bar() {}
```

```
// app.js
import bar from ‘./bar’;
```

ES6 Module Example

```
// foobar.js
var foo = function() {}
export default foo;
export function bar () {}

// app.js
import {foo, bar} from ‘./foobar.js’;

import {bar as magic} from ‘./foobar.js’;

import * as lib from ‘./foobar.js’;
```

Modules for the Web

- **Compile it**
  - Webpack
    - Built for complex work pipelines & transformations
    - Supports CommonJS and AMD modules
    - https://webpack.js.org/
  - Browserify
    - Built to parse node-like modules
    - http://browserify.org/
- **Transpile it**
  - Babel
- **Load it**
  - System.js
    - Universal module loader
  - RequireJS
    - AMD Modules

Modules – Recap

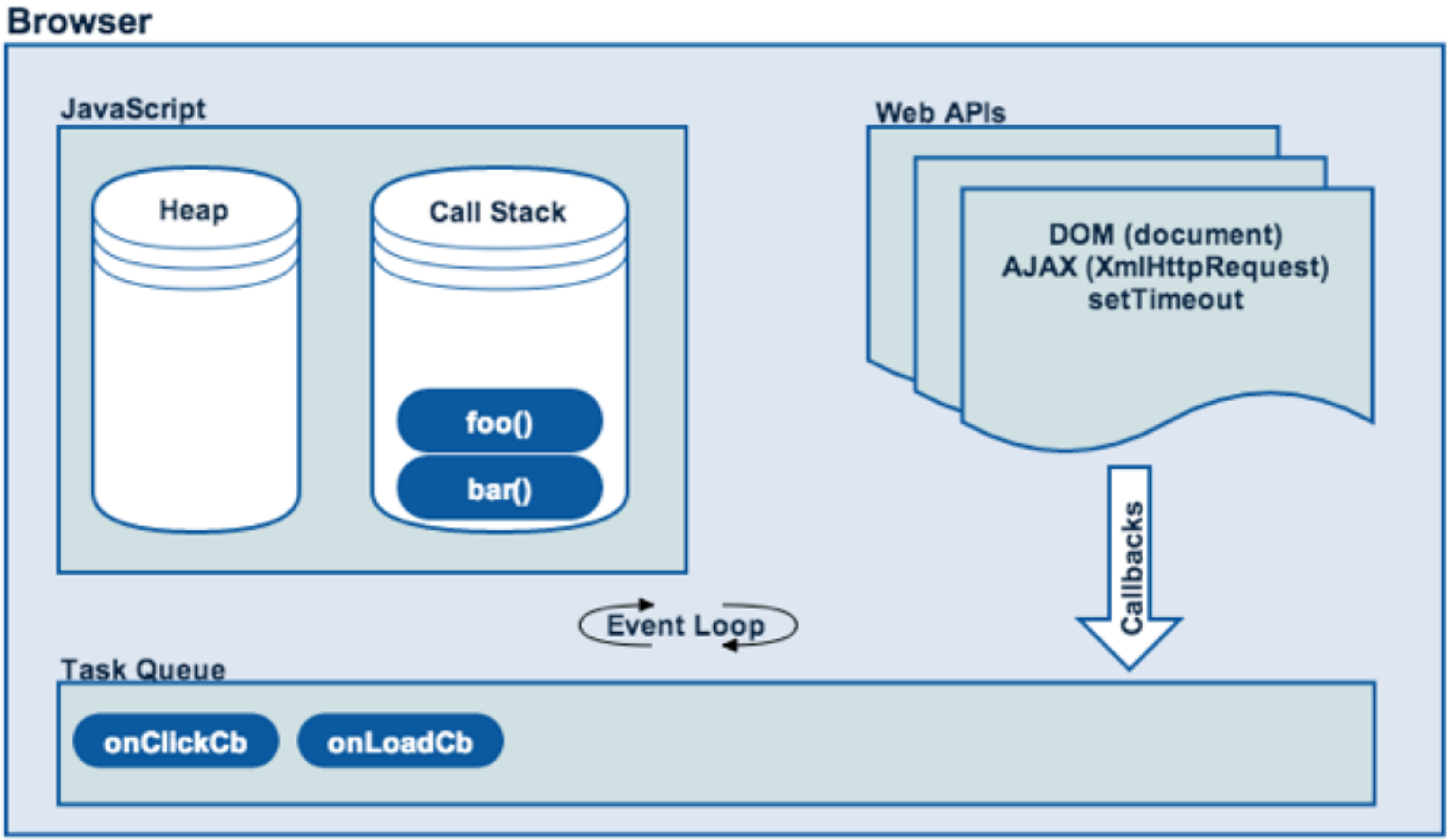
- ~~No current language-level support for modules~~
  - ES6 Support!
- Revealing Module Pattern, which is an IIFE, can solve simple problems
- AMD, via require.js, to manage more complex or on-demand modules and their dependencies
- ES6 will make all this obsolete, whenever browser vendors decide to support it fully

module

ASYNCHRONOUS PROGRAMMING

Single-threaded JavaScript

- Does everyone know the event-loop?



Being Asynchronous

- Because JavaScript cannot do more than one thing at a time…
  - Callbacks
  - Promises
  - ES6 - `async` and `await`

Callback Pattern

- A function that is passed to another function as a parameter, so that it can be invoked later by the calling function.
- They aren’t asynchronous by themselves (they tend to be used for asynchronous operations, however)
- Event listeners, ajax handling, file and network requests

```
function callLater(fn) {
  // do some async work
  return fn();
}

callLater(function() {
  console.log(“I’m done!”);
});
```

Callback Context

- `this` inside a callback may change, be careful

```
setTimeout(function() {
    console.log(“I was called later”);
}, 1000);

$(‘a’).on(‘click’, function() {
    console.log(this); // ?
});
```

The Downside to Callbacks

- Can become deeply nested and not easy to reason
- There is no guarantee that the callback will be invoked when you expect, if at all

```
// callback hell
async1(function(err, result1) {
   async2(function(err, result2) {
        async3(function(err, result3) {
             async4(function(err, result4) {
                   /*…*/
             });
        });
   });
});
```

Enter, Promises

- A **Promise** represents a proxy for a value not necessarily known when the promise is created
  - They represent the *promise* of having a value at some point in the future, instead of the final value.
- Benefits
  - Guarantees that callbacks are invoked
  - Composable (can be chained)
  - Immutable (one-way latch)
  - You can continue to use them after resolved
- Bummers
  - ES6
  - No `.finally()`

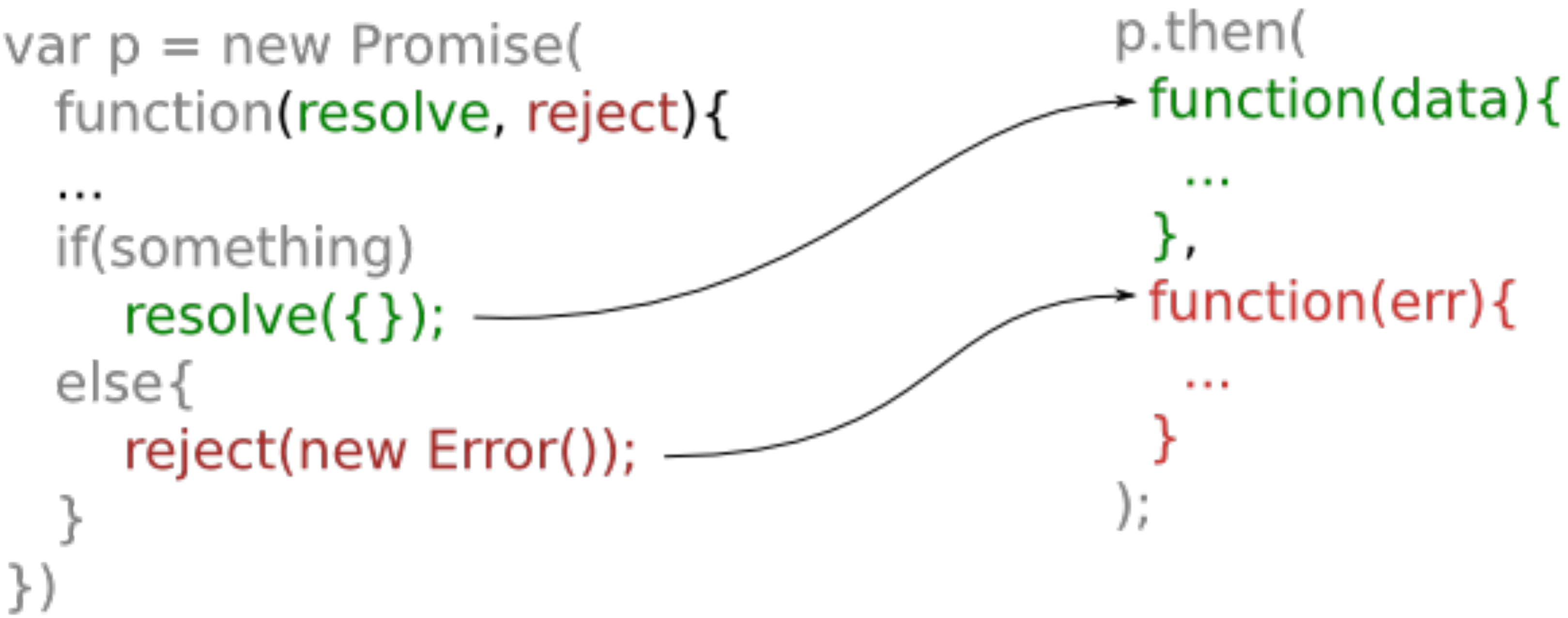
Making a Promise

- Construct a Promise to represent a future value
  - Constructor expects a single argument, which is a function that has two arguments, **`fulfill`** and **`reject`**
- Attach handlers using **`then`** method
  - The handler consumes the later-value when it’s ready
  - And handles errors, too

```
var promise1 = new Promise(function(fulfill, reject) {
        async1(function(err, data) {
                if (err) {
                        reject(err);
                } else {
                        fulfill(data);
                }
        });
});
promise.then(onFulfilled, onRejected);
```

Promises Terminology

- Specification: https://promisesaplus.com
  - **pending** – the action is not fulfilled or rejected
  - **fulfilled** – the action succeeded
  - **rejected** – the action failed
  - **settled** – the action is fulfilled or rejected



```
var p = new Promise(
  function(resolve, reject){
  ...
  if(something)
    resolve({});
  else{
    reject(new Error());
  }
})
```

```
p.then(
  function(data){
    ...
  },
  function(err){
    ...
  }
);
```

Promise Errors

- Use the reject/error handler argument in `then()` to handle when an Exception is thrown from the *current* promise being handled
- ES6 Promises also support a `.catch()` callback, which will do the same thing.

```
var promise1 = new Promise(function(fulfill, reject) {
     setTimeout(function() {
         reject(“Something went wrong!”);
     }, 1000:
});


promise1.then(null, function(error){
     console.log(‘Something went wrong’, error);
});

prom1.catch(function(err) {
    console.log(err);
});
```

Chaining Promises

- **`.then()`** wraps all return values in a new Promise, allowing us to chain our then()s
- You can return a new promise to create sequences that depend on the previous step

```
var promise1 = new Promise(function(fulfill, reject) {
     setTimeout(function() {
         fulfill(5);
     }, 1000:
});

promise1.then(function(data){
    console.log(data); // 5
    return data + 2;
}).then(function(data) {
    console.log(data); // ?
}).catch(function(err) {
    console.log(err);
});
```

Fixing callback hell

- Remember this? Let's see what that would look like if we wrapped each async operation in a promise

```
async1(function(err, result1) {
    async2(function(err, result2) {
        async3(function(err, result3) {
        });
    });
});
```

Promised Land

- If each of our async functions returned a promise object, we could do this:

```
promise1
    .then(promise2)
    .then(promise3)
    .catch(function(err) {
        // deal with thrown error
    });
```

Promise breaking

- What is wrong with the below promise sequence?

```
fetchResult(query)
    .then(function(result) {
        // this is an async operation
        asyncRequest(result.id);
    })
    .then(function(newData) {
        console.log(newData);
    });
    .catch(function(error) {
        console.error(error);
    });
```

Composing Promises

- `Promise.all([…])`
  - Returns a promise that resolves when all promises passed in are resolved or at the first rejection
  - Fulfilled value is an array of all returned promise values
- `Promise.race([…])`
  - Returns a promise that resolves when any one promise is fulfilled or rejected

Composing Promises Example

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 1000);
});

Promise.all([p1,p2,p3]).then(function(data) {
    console.log(values); // ?
});

Promise.any([p1,p2,p3]).then(function(data) {
    console.log(data); // ?
});
```

Async and await

- Two new keywords needed to write asynchronous code that looks and feels synchronous
- **`async`**
  - Defines an AsyncFunction that can yield flow of control
  - AsyncFunction returns a promise that will be resolved when the function returns a value or rejected when it has an Error
- **`await`**
  - Informs code within an async function to yield/wait for the promise to complete before proceeding
  - Must be inside an “async” function… careful with anonymous functions

Async/await Example

```
// must define our function as asynchronous
async function getAndDoSomething() {

  // await yields flow of control back to caller
  // until the promise is settled
  var artist = await Ajax.get("/api/artists/1");
  artist.albums = await Ajax.get(
    “/api/artists/1/albums"
  );

  View.set("artist", artist);
}

var prom1 = getAndDoSomething();
```

Exercise - Promises

1. Open the following JavaScript file:

```
www/promises/promises.js
```

2. Follow the instructions inside the file
3. Test it by running it

```
node www/promises/promises.js
```

module

TESTING

Testing in the Browser

- In order to achieve comprehensive testing in JavaScript you need to:
  - Test your code in the browser
  - Test it in every browser you support
  - Use a tool to automate this

Different levels of testing

- Unit Testing
  - Low-level test for a small piece of code
- Integration Testing
  - Demonstrates different units of the system working together
- End-to-end Testing
  - Aka Functional, System, or Acceptance
  - The system is run in full and nothing is faked

Testing levels: A metaphor

- Think of a Bike as a system*
  - A **unit test** might verify the *brake wire is sturdy* or that *the brake handle exerts pressure*.
  - An **integration test** would test to ensure that *when the brake handles are used the tires get x lbs. of pressure*
  - **End-to-end** testing will combine it all in a real environment: “*Braking on a tar road while riding at 15mph should stop in 15 meters*”

Styles of writing tests

- Classic Unit tests:

```
assert("empty objects", objects.length > 0);
```

- Specification tests

```
expect(objects.length).toBeGreaterThan(0);
```

- The difference is in how you write it
  - Test names are written as sentences, which are more expressive, with “Should” as a common prefix

Jasmine - our test framework

- One of many test frameworks
- Specification-based testing
- Expectations instead of assertions
- The documentation is great
    - https://jasmine.github.io/2.0/introduction.html

Example: Writing Jasmine Tests

```
describe("ES6 String Methods", function() {
  it("has a find method", function() {
    expect("foo".find).toBeDefined();
  });
});
```

Basic Expectation Matchers

- toBe(x): Compares with 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): Math.abs(e - actual) < (Math.pow(10, -p) / 2)

Smart Expectation Matchers

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

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.

Deferred Tests

- Tests can be marked as pending, so you can write it but run it later

```
it("declared without a body!");

//or:

it("uses the pending function", function() {
  expect(0).toBe(1);
  pending("this isn't working yet!");
});
```

Spies

- Stub a function and track calls and arguments
- Removed after each spec, so run in your spec or in beforeEach()

```
// basic set up
describe(“foo”, function() {
  var foo;

  beforeEach(function() {
    foo = {
      add: function(n) {return n + 1;},
    };
  });

});
```

Example: Spying

```
it(“should be called”, function() {
  spyOn(foo, ‘add’);
  foo.add(1);
  expect(foo.add).toHaveBeenCalled();
  expect(foo.add).toHaveBeenCalledWith(1);
  expect(foo.add.calls.count()).toEqual(1);
  expect(x).toBeUndefined(); // no return!
});
```

Example: Spying and Faking

```
spyOn(foo, “add”).and.callThrough();

spyOn(foo, “add").and.returnValue(745);

spyOn(foo, “add")
  .and.callFake(function(any, arg) {
    return 1001;
  });

spyOn(foo, “add”).and.throwError("eek!");
```

Mocking Timeouts (setup)

```
var timedFn;

beforeEach(function() {
  // createSpy creates a bare function to spy on
  timedFunction = jasmine.createSpy("timedFn");
  jasmine.clock().install();
});

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

Mocking Timeouts (testing)

```
it(“should be called after 100 ms", function() {
  // act
  setTimeout(function() {
    timedFn();
  }, 100);

  // The callback shouldn't have been called yet:
  expect(timedFn).not.toHaveBeenCalled();

  // Move the clock forward and trigger timeout:
  jasmine.clock().tick(101);

  // Now it's been called:
  expect(timedFn).toHaveBeenCalled();
});
```

Mocking Intervals

```
it(“should be called twice in 200ms", function() {
  // act
  setInterval(function() {
    timedFn();
  }, 100);

  // The callback shouldn't have been called yet:
  expect(timedFn).not.toHaveBeenCalled();

  jasmine.clock().tick(101);
  expect(timedFn.calls.count()).toEqual(1);

  jasmine.clock().tick(100);
  expect(timedFn.calls.count()).toEqual(2);
});
```

Testing Asynchronous Functions

```
describe("async function testing", function() {

  it(“should take a while", function(done) {

    setTimeout(function() {
      done(); // tell Jasmine we were called.
    }, 2000);

  });

});
```

Running Jasmine Tests

- Standalone runner:
  - List files in SpecRunner.html
    - try it out: http://localhost:3000/jasmine/SpecRunner.html
  - Opening that file in your browser runs the tests
- Node.js runner:
  - Provides a jasmine tool
  - Runs tests inside Node.js
- Karma-Jasmine runner:
  - Automatically manages browser farms
  - Runs tests in parallel on all browsers
  - Can use headless browsers (PhantomJS)
  - Support for continuous integration

Best Practices for Testing

- Setup, Execute, Expect (or Arrange, Act, Assert)
- Keep tests independent
- Don’t change global state
- Don’t test implementation
  - Avoid privates
- Make sure your tests actually fail
- Avoid testing DOM specifics

Further Information

- Frameworks
  - Mocha, Jasmin, qUnit
- Runners
  - Karma (client-side), Protractor (end-to-end)
- Headless Browsers
  - Zombie, PhantomJS
- Assertion Libs
  - Sinon.js
    - Advanced fake/stub/mock lib
    - Fake a server, ajax
- Mocking Libs
  - Advanced assertion lib

class project

MVC

The Basic Ideas Behind MVC

- Design pattern for building complex apps
- Divide application functionality into (at least) three layers
  - **Model**: manages data and business logic independent of the user interface
  - **View**: Display something about the model to the user
  - **Controller**: Receives user input and facilitates decoupled communication between the view and the model

Web Applications and MVC

- MVC has been widely adopted and extended in the web world
- Several JS frameworks support MVC (AngularJS, Ember.js, Backbone, etc.)
- Modern browsers allow the entire MVC stack to exist entirely in JS
- Typical applications use Ajax+JSON+REST and a back-end database

Objectives for the Class Project

- Simple, single page application (one HTML file)
- Uses MVC with a back-end JSON server
- Pure JavaScript (no frameworks)
- Features:
  - Display a list of musical artists
  - Clicking an artist shows their albums
  - Ability to add a new artist

Things we're going to need

- Simple wrapper around Ajax (we'll write this ourselves)
- Template library (we'll use Mustache)
- Testing framework (we'll use Jasmine)

Ajax Refresher

- Maxing an Ajax request:

```
var 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);
```

Exercise - A Simple Ajax Library

1. Open `www/discography/js/lib/ajax.js`
2. Fill in the missing pieces
3. Run the tests:

   ```
   node bin/jasmine spec/ajax.spec.js
   ```
4. Get the tests to pass

What a Model Should Provide

- Fetch all records (within reason)
- Fetch a single record by ID
- Create a new record and send to back end
- Update a record and save to back end
- Delete a record from the back end
- Contain any business logic

Using REST + JSON

- Fetch all artists (no body):
  - GET /api/artists
- Fetch a single artist (no body):
  - GET /api/artists/2
- Create a new record (JSON body):
  - POST /api/artists
- Update a record (JSON body):
  - PATCH /api/artists/2
- Delete a record (no body):
  - DELETE /api/artists/2

Exercise - Our First Model

1. Open
   `www/discography/js/models/artist.js`
2. Fill in the missing pieces
3. Test with the following command:

   ```
   node bin/jasmine
   ```
4. Play with the code in the browser console

A Word about Mocked XHR

- Our tests use a custom Ajax spy/mocking library
- Tell the library you want to hijack Ajax calls:

```
var artist = {name: “The Wombats”};
ajaxSpy(‘get’, artist);

/* call a function that uses the ‘Ajax’ module */
```

Exercise - Adding Model Tests

1. Open
   ```
   spec/artist.spec.js
   ```
2. Add a test for the fetchAll function
3. Add a test for the save function
4. Add a test for the destroy function
5. Test with the following command:
   ```
   node bin/jasmine spec/artist.spec.js
   ```

Displaying Artists in the Front-end

- Let’s get our list of artists into the HTML:
  - We’ll display them in an HTML table
  - To do this we’ll need a view library
  - And eventually, a controller to load the artists and send them to the view

Cheating with the Mustache Library

- The mustache library makes it easy to create view templates:
  - Given a template:

    ```
    Hello {{name}}
    ```

  - And an object:

    ```
    {name: “World”}
    ```

  - Mustache produces:

    ```
    “Hello World”
    ```

Implicit Loops with Mustache

- Given a template:

```
{{#friends}}
  <li>{{name}}</li>
{{/friends}}
```

- And an object:

```
{friends: [
  {name: “Moss”},
  {name: “Roy”}
]}
```

- Mustache produces

```
“<li>Moss</li>\n<li>Roy</li>”
```

Putting Mustache Templates in our HTML

- We can put mustache templates directly in our HTML

```
<script id="my-template" type="x-tmpl-mustache">
Hello {{name}}!
</script>
```

- And fetch them when needed:

```
var obj = {name: "World"};
var tpl = document.getElementById("my-template");
var out = Mustache.render(tpl.innerHTML, obj);
```

Exercise - Create the View Object

1. Open `www/discography/js/lib/view.js`
2. Fill in the missing pieces
3. Test the `render` method in the browser console

A Simple Controller

- To glue everything together we’re going to need a controller.
- It should:
  - Fetch all artists
  - Use the View object to render the view
- We’ll call this the *index* action

Exercise - Build a Simple Controller

1. Open `www/discography/index.html`
2. Create a mustache template for the artist index view
3. Open `www/discography/js/controllers/artists_controller.js`
4. Fill in the `index` function
    1. Fetch all artists
    2. Render the view
    3. Insert the view into the #view div
5. Reload the page in the browser

Bonus Exercise

- Turn artist names into links
  1. Clicking a link should invoke the ArtistsController.show method and display a single artist
  2. Fill in the remaining functions in the ArtistsController
- Hints
  - On the table row, store a data-artist-id attribute with the ID of the current artist. You can use this in the click event callback to send the ID to the AristsController.show method

Abstracting Our Model Code

- IF we wrote another model it would look very similar to the Artist model. It would have:
  - The ability to create a new record
  - Save a record
  - Delete a record
- Let’s fix that

Exercise - Factoring Out Common Functionality

1. Open `www/discography/js/lib/model.js`
2. Move all common logic from Artist into Model
3. Link the Artist and Model objects (prototype?)

Nested REST Resources

- Fetch all albums for artist 2 (no body):
  - GET /api/artists/2/albums
- Fetch a single album for artist 2 (no body):
  - GET /api/artists/2/albums/3
- Create a new record (JSON body):
  - POST /api/artists/2/albums
- Update a record (JSON body):
  - PATCH /api/artists/2/albums/3
- Delete a record (no body):
  - DELETE /api/artists/2/albums/3

Exercise - The Albums Model

1. Open `www/discography/js/models/album.js`
2. Fill in the Album model
    1. Ensure that albums have an artist_id property
    2. Make sure all Ajax requests go through /api/artists/{n}/albums
3. Test in the browser console

Exercise - The Albums Controller

1. Open `www/discography/js/controllers/albums_controller.js`
2. Update the controller and views so that clicking an artist will display a list of albums

Exercise - Creating New Artists

1. In the artists view, add a link for creating a new artist
2. Clicking the link should display a form
3. Submitting the form should save an artist to the database
4. Then display all artists again

module

ES6

ECMAScript 6

- ~~ECMAScript 2015~~
- Lots of syntax sugar and a few wishlist features
  - let/const
  - fat arrow functions
  - class keyword
  - Promises
  - Iterators
  - Generators
- Standardizing objects
- Cleaning up the global scope

`let` keyword

- Uses block scope!
- Does not hoist (Temporal Dead Zone)

```
if (expression) {
  var a = 1; // scoped to wrapping function
  let b = 2; // scoped to the block
}

console.log(a); // 1
console.log(b); // undefined
```

`let` keyword in for-loops

- Using let with a for loop in ES6 is better than var

```
for (let i=0; i<5; i++) {
  // i is scoped each iteration
  setTimeout(function() {
    console.log(i);
  }, 1000);
}

// outputs
// 0, 1, 2, 3, 4
```

`const` keyword

- Immutable
- Uses block scope
- Does not hoist either

```
{
  const a = 1;
  const bla = {name: ‘Ryan’}

  bla.name = ‘Tim’; // ok, objects mutable
  a = 10; // TypeError
}
```

Arrow Functions

- (Fat) Arrow functions support super-short syntax in a few ways
- No own **`arguments`** variable
  - it's the arguments of the outer function
- No own **`this`**
  - Lexically bound

```
var add = function (x) {
  return x + 1;
}

// becomes
var add = x => x + 1;
```

Arrow functions continued

```
var add = function (x, y) {
  return x + y;
}

// becomes
var add = (x, y) => x + y;

// which is also
var add = (x, y) => {
  return x + y; // what is this here?
}
```

Object Shorthand

- Can omit the **`:value`** portion in object definition when there is a var with the same key name

```
var name = ‘Ryan’;
var doSomething = function() {};

var ryanInterface = {
  name,
  doSomething
}
```

Object Method Definition

- Functions can be defined w/out a explicit key name (it will be the function name)

```
var name = ‘Ryan’;

var ryanInterface = {
  name,
  doSomething() {
    console.log(‘Hi!’);
  }
}
```

Object Destructuring

```
var ryanInterface = {
  name: ‘Ryan’,
  other: 1,
  deep: {
    color: ‘Red’
  }
}

var {name} = ryanInterface;
var {name, other} = ryanInterface;

// rename to ryanName
var {name : ryanName} = ryanInterface;
var {deep: {color}} = ryanInterface;
```

Array Destructuring

```
var colors = [‘red’, ‘orange’, ‘white’];

var [red, orange] = colors;
console.log(red);
console.log(orange);

var [, , white] = colors;
console.log(white);
```

Function Parameter Defaults (!)

```
function addTo(base, add=1) {
  return base + add;
}
```

Rest parameter

- Rest is a special parameter in functions that captures all remaining parameters passed in

```
// replacement for arguments
function sumAll(first, …remaining) {
  return first + remaining.reduce(total, next) {
    return total + next;
  }
}
// while destructuring
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }
```

Spread operator

- Expands any iterable object into standalone values

```
// combining arrays
var arr1 = [1,2,3];
var arr2 = [4,5,6];
arr1.push(…arr2);
var arr3 = [1,2,3,…arr2,7,8,9];

// using an array as individual arguments
function expectsThree(a, b, c){};
expectsThree(…arr1);
```

Class keyword

- Just syntactic sugar over prototypes
- Leaky abstraction; you’ll still deal with prototypes
- Not hoisted (like function declarations are)

```
var Rectangle = class {}

class Square extends Rectangle {
  constructor (width, color) {
    super(width, width);
    this.color = color;
  }
  someMethod() {
    return “Hi”;
  }
}
```

`Class` keyword extras

- You can class-extend traditional function-based “classes”
- Can define static methods with static keyword on the method function
  - Won’t be created on instances
- Can define getters and setters with get and set method keywords

Generic `for` loop

- `for…of`
  - works with any **iterable** collection object
  - Arrays, Strings, Maps (not Objects)
  - `for…in` is for object properties that are enumerable, `for…of` is for collections

```
var iterableCollection = [1,2,3];

for (let value of iterableCollection) {
  console.log(value);
}
```

Iterators

- Object that knows how to access items from a collection one at a time

```
let something = {
  [Symbol.iterator]: function() {
    let n = 0;
    return {
      next: () => ({value: n, done: n++ >= 10})
    };
  }
};

for (let x of something) {
  console.log(x);
}
```

Generators

- Allow a function to generate many values over time by returning an object which can be iterated over
- They can **`yield`** and retain context until completion
- You can pass values in
- A return statement indicates done

```
function* idMaker() {
  yield 2;
  yield 5;
  return 10;
}

var gen = idMaker();
gen.next(); // {value: 1, done: false}
gen.next()
```

Maps

- Object that holds key-value pairs
- Can use objects as keys, which is interesting

```
let characters = new Map();

characters.set("Ripley", "Alien");
characters.set("Watney", "The Martian”);

characters.has("Ripley"); // true
characters.get("Ripley"); // “Alien"

characters.keys();
characters.values();
```

WeakMaps

- Like a Map, but keys can be garbage collected
  - Keys must be objects
  - Could be useful to hold references to DOM elements but allows garbage collection once they are removed from the page
- Same API as Map

Async and await

- `async`
  - Defines functions that can yield flow of control
  - They always return a promise
- `await`
  - Informs code within an async function to yield/wait for the promise to complete before proceeding
- With these two keywords you can write asynchronous code that looks and feels synchronous

From this…

```
function getAndRenderArtists() {
  var artists;
  Ajax.get(“/api/artists/1”)
   .then(function(data){
      artists = data;
      return Ajax.get(“albums”);
    })
    .then(function(data){
      artists.albums = data;
      View.set("artist", artist);
    })
    .catch(function(err){});
}
```

… to this

```
async function getAndRenderArtists() {
  var artist = await Ajax.get("/api/artists/1");
  artist.albums = await Ajax.get(
    “/api/artists/1/albums"
  );
  View.set("artist", artist);
}
```

And more…

- Set and WeakSet
  - Collections of values (no keys)
  - One occurrence per value
- Proxy and Reflect
  - Powerful objects for meta-programming.
- Symbol
  - Create and use runtime unique entries in the symbol table.
- Template Literals
  - String interpolation: `` `Hello ${name}` ``

ES6 Support

- Depends on what you want to do…
  - http://caniuse.com/#search=es6
- Transpile it
  - Babel
- Node?
  - http://node.green/
  - No module support

module

WEB APIS

Web Storage

- 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");
var item = sessionStorage.getItem("key");
sessionStorage.removeItem("key");
```

Local Storage

- Lifetime: unlimited
- Sharing: All code from the same domain
- 5MB user-changeable limit (10MB in IE)
- Basic API:

```
localStorage.setItem("key", "value");
var 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 Storage - Browser Support

- IE >= 8
- Firefox >= 2
- Safari >= 4
- Chrome >= 4
- Opera >= 10.50

Web Storage - Documentation

- https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage
- https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

AppCache

- A server-side manifest file
- Tells the browser which files to long-term cache
- Allows a web site to work offline

Example Manifest File

- Add the manifest attribute to your HTML:

```
<html manifest="/site.appcache">
<!-- ... -->
</html>
```

- Create the manifest file on your server:

```
CACHE MANIFEST

CACHE:
/favicon.ico
index.html
app.js
app.css

NETWORK:
*
```

AppCache Server Side Requirements

- The server must transmit the manifest file with the Content-Type set to text/cache-manifest
- The server should send the correct cache and E-Tag headers to the browser to keep the browser from caching the manifest file too long
- The manifest file should be generated server-side with comments in the file containing the E-Tag headers for each listed file

AppCache Client Side Requirements

- Once you start using application caching the cache becomes the default source for all requests
- The browser will use the application cache even if the user is online
- The browser won’t allow network traffic back to the site for uncached resources by default
- Make sure your manifest has a NETWORK: section with *

Updating the Cache in Long-lived Applications

- Periodically (once a day) call update: applicationCache.update();
- Listen for update events and notify the user

```
(function(cache) {
  cache.addEventListener(
    'updateready',
    function() {
      if (cache.status === cache.UPDATEREADY) {
      // Tell the user to reload the page.
    }
  });
})(applicationCache);
```

App Cache - Browser Support

- IE >= 10
- Firefox >= 3.5
- Safari >= 4
- Chrome >= 4
- Opera >= 11.5

Canvas: Two Drawing APIs

- 2D drawing primitives via paths
- 3D drawing via WebGL
- Both can be hardware accelerated
- Typically 60 FPS (if animating)

Drawing a Circle: The HTML

```
// index.html
<canvas id="circle"></canvas>
```

```
// app.js
canvas = document.getElementById("circle");
context = canvas.getContext("2d");
var path = new Path2D();
path.arc(75, 75, 50, 0, Math.PI * 2, true);
context.stroke(path);
```

Canvas - Browser Support

- IE >= 9
- Firefox >= 1.5
- Safari >= 2
- Chrome >= 1
- Opera >= 9

File API

- It’s not a general-purpose I/O interface
- It only lets you get basic info about user-selected files:
  - Name
  - Size
  - MIME type
- A user selects a file with an <input> or using drag and drop

Example - File Size

- In the HTML:

```
<input type="file" id="the-input">
```

- In the JavaScript (after the user picks a file):

```
var input = document.getElementById(
 "the-input"
);
var size = input.files[0].size;
```

File API - Browser Support

- IE >= 10
- Firefox >= 3.0
- Safari >= 6.0
- Chrome >= 13
- Opera >= 11.5

Geolocation

- User can share their location (based on ip or device)
    - Can opt in/out
- Get long/latitude data
- “geolocation” object

```
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(
         successFunc(pos) {
              pos.coords.latitude;
              pos.coords.longitude
         },
         failFunc(posError){…}
    );
};
```

Geolocation API

- Geolocation Object
  - Request through here
  - Child of navigator
  - getCurrentPosition(successCallback, failCallback);
- Position Object
  - Position.coords.latitude
  - Position.coords.longitude
  - Accuracy (in meters)
  - Altitude
  - Heading, speed, timestamp
- PositionError Object
  - Code and a message

Geolocation Examples

- Basic
  - http://jsfiddle.net/mrmorris/nqNxU/
- Little more to it
  - http://jsfiddle.net/mrmorris/s4xtduo2/

Geolocation - Browser Support

- IE >= 9
- Firefox >= 3.5
- Safari >= 5
- Chrome >= 5
- Opera >= 16

Web Workers

- Allows you to start a new background “thread”
- Messages can be sent to and from the worker
- Message handling is done through events
- Load scripts with: importScripts("name.js");

Web Workers - Browser Support

- IE >= 10
- Firefox >= 3.5
- Safari >= 4
- Chrome >= 4
- Opera >= 10.6

Web Sockets

- Full duplex connection to a server
- Create your own protocol on top of WebSockets
- *or* use an existing library and protocol
- Not subject to the same origin policy (SOP) or CORS

Web Sockets - How it works

- The browser requests that a new HTTP connection be upgraded to a raw TCP/IP connection
- The server responds with HTTP/1.1 101 Switching Protocols
- A simple binary protocol is used to support bi-directional communications between the client and server over the upgraded port 80 connection

Web Sockets - Security Considerations

- There are no host restrictions on WebSockets connections
- Encrypt traffic and confirm identity when using WebSockets
- Never allow foreign JavaScript to execute in a user’s browser

Web Sockets - Browser Support

- IE >= 10
- Firefox >= 6
- Safari >= 6
- Chrome >= 14
- Opera >= 12.10

Server-sent events

- Pros:
  - Simpler than WebSockets
  - One direction: server to browser
  - Uses HTTP, no need for a custom protocol
- Cons:
  - Not supported in IE (any version)
  - Poor browser support in general (polyfills are available)
- How:
  - Browser: use the EventSource global object
  - Server: just write messages to the HTTP connection

module

LAY OF THE LAND

JavaScript lay of the land

- Popular libraries
- Popular frameworks
- Toolchain(s)
- Keeping it all straight

Popularity

- jQuery
  - Thin wrapper over the browser API (not really a framework)
- Backbone.js
  - Model-view-presenter (MVP) library
- Ember.js
  - Model-view-controller (MVC) framework for single-page applications
- Angular.js
  - Model-view-viewmodel (MVVM) framework for single-page applications
- React.js
  - View-layer framework for fast, DOM-free views

jQuery

- A thin wrapper over the standard browser API
- Universal APIs for all browser
- Entire API exported via the $ function

jQuery Example

```
$("#the-input").on("keydown", function() {
  $("#the-output").text($(this).val());
});
```

Backbone.js

- A thin wrapper over jQuery and Underscore
- Provides prototypes for models and views
- Works with a back-end JSON+REST server
- Doesn’t impose any structure on your application

Backbone.js Example

- Please see the demo application in the provided zip file.
- If your node server is running you can use this link:
  - http://localhost:3000/frameworks/backbone/

Ember.js

- Complete framework for single-page applications
- Compels you to structure you application in a specific way
- Requires jQuery and Handlebars libraries
- Includes support for:
  - URL routing
  - Models and controllers
  - View templates using the handlebars library

Ember.js Example

- Please see the demo application in the provided zip file.
- If your node server is running you can use this link:
  - http://localhost:3000/frameworks/ember/

AngularJS

- Complete framework for single-page applications
- Compels you to structure you application in a specific way
- Standalone library with no external dependencies
- Includes support for:
  - URL routing to specific controllers
  - Models, Controllers, and HTML views
  - Designed for application testing
  - A variety of third-party plugins

AngularJS Example

- Please see the demo application in the provided zip file.
- If your node server is running you can use this link:
  - http://localhost:3000/frameworks/angular/

React.js

- View-only library (you still need models and Ajax)
- Removes direct DOM manipulation
- Fast, efficient HTML rendering into the DOM
- Created and maintained by Facebook

AngularJS Example

- Please see the demo application in the provided zip file.
- If your node server is running you can use this link:
  - http://localhost:3000/frameworks/react/

Languages that compile to JavaScript

- PureScript
- Flow
- TypeScript
- Dart
- CoffeeScript

Transpilers

- Convert between JS versions
- Popular Options
  - Babel
  - Traceur
  - Browserify

Node.js

- Server-side JavaScript engine
- Also provides a general-purpose environment
- Write servers, or GUI programs in JavaScript
- Most development tools are written in JavaScript and use Node.
- https://nodejs.org/

Node Package Manager (npm)

- Repository of JavaScript libraries, frameworks, and tools
- Tool to create or install packages
- Run scripts or build processes
- 250k+ packages available
- https://www.npmjs.com/

Toolchain

- JavaScript build/automation tools that:
  - Transpile or generate JavaScript as necessary
  - Combine all JavaScript files into a single file
  - Minify and compress JavaScript (easy to deploy)
- Popular options
  - Grunt
  - Gulp
  - Broccoli

Toolchain - JSHint and ESLint

- Linting tools
- Suggests changes to your JavaScript
- ESLint is fully configurable, easy to add custom rules
- Enforce project style guidelines

Babel

- Automated JavaScript restructuring, refactoring, and rewriting
- Parses JavaScript into an Abstract Syntax Tree (AST)
- The AST can be manipulated by scripts written in JavaScript
- Presents include:
  - ES6 to ES5 transpiling
  - JSX to JavaScript conversion
  - And tons more. . .

that's a wrap

QUESTIONS?