learning spike

Intermediate JavaScript

Ryan Morris
@mrmorris

Introductions

- About me…
- About you…
    - Name?
    - What do you do here?
    - What is your programming background?
        - Any front-end?
        - 😍, 😡 or 😭 JavaScript?
    - What do you hope to gain from this course?

How the class works

- Lecture & labs
- Informal
- Flexible outline
  - You help me define areas of interest
  - Too much to cover!
- Exposure to *Intermediate JS* concepts
- Class review at the end of the day

Get the most out of the class

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

What we'll cover

- A little review (js, html, css, dom)
- Ajax/XHR
- Built-in Objects
- Objects in-depth
- Prototype & Inheritance
- Asynchronous JS (Promises)
- Observables?

I wasn't planning to cover
- The Basics
- jQuery (incl. Ajax)
- ES6 in depth
- Modules

~Mostly ES5~

~Mostly for intermediates~

~You should be familiar with js, html, css~

Resources

- Reading List
  - https://javascript.info/intro
  - You Don’t Know JS
    - https://github.com/getify/You-Dont-Know-JS
- Documentation
  - http://devdocs.io
  - https://developer.mozilla.org/en-US/docs/Web
  - Google it.
- Compatibility checks
  - http://caniuse.com

Set up

- A browser with dev tools
  - Preference for Chrome in class
  - Open your browser and hit `F12` or `alt/opt/⌥ -⌘-i`
- Sign up with jsFiddle.net
  - http://jsfiddle.net/
  - Does this work?
    - http://jsfiddle.net/mrmorris/8wfu5tct/
    - You should see “We are ok!” message

👆 Everyone OK with the above?

jsFiddle primer

- It’s a **sandbox**
- It’s a set of **iframes**
  - Check which frame you’re accessing via your console
- It runs in an **IIFE** unless you ask it not to
  - So your stuff isn’t global…
- When you start a lab…
  - Fork it (copy) — you’ll own that!
  - “*update*” to save!
  - “*run*” to test!
  - “*set as base*” to make a version the *main* version

refresher…?

DEBUGGING

Browser Debugging

- Use browser dev tools to access its JavaScript console
  - The browser's "`console`" is a REPL
  - log output for testing
- Can also use dev tools to:
  - set breakpoints & debug js
  - view network requests
  - view memory usage
  - inspect html + css

The `console` object

- Console api
  - `console.log(); // echo/print/output`
  - `console.assert(); // test`
  - `debugger; // breakpoint`
- Gotchas
  - Console methods are asynchronous
    - They may not run in the order you expect
  - They are not available in every browser
- Seeing a bug/issue?
  - Clear your console of old errors
  - Check where the error happened

refresher

JAVASCRIPT

What is JavaScript?

- Standardized as **ECMAScript**
- **Interpreted**
- Case-sensitive C-style syntax
- Dynamically typed (with weak typing)
- Fully **dynamic**
- **Single-threaded** event loop
- Unicode (UTF-16, to be exact)
- **Prototype**-based (vs. class-based)
- Kind of weird but enjoyable

JavaScript Versions

- ES3/1.5
  - Released in 1999 – in all browsers by 2011
  - IE6-8
- **ES5/1.8**
  - Released in 2009
  - IE9+
  - http://kangax.github.io/compat-table/es5/
- **ES6 [EcmaScript 2015] mostly supported**
- ES7 [EcmaScript 2016] finalized, but weak support
- ES8 [EcmaScript 2017] finalized in June 2017
- ES.Next…

Basic constructs

- **We should be OK with:**
  - variables
  - data structures like arrays or maps
  - if-else statements
  - for and while loops
  - functions

Core JS concepts

- **We should be OK on:**
  - Data Types
    - Objects, Functions, Arrays
  - Coercion
  - Scope & Hoisting
  - Object literals
  - Function declaration vs expression
  - Context (***this*** keyword)

If we're not OK on a topic here we *should* dive into it

Refresher - Data Types

- There are 5 primitives (string, number, boolean, null, undefined) and then Objects
  - Functions are a callable Object
  - Objects are maps of properties referencing data
  - Arrays are for sequential data
- Declare variables with "`var`"
  - **Function scope**
  - Block scope in ES6 with "`let`" and "`const`"
- 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

Refresher - Type Coercion

- If a variable type is not what JavaScript expects, it will convert it on the fly, based upon the context

```
var x = “ryan”; // a literal
“ryan”.length; // is coerced to a… ?


+"42"; // 42
"Name: " + 42; // "Name: 42"
1 + "3"; // 4;
“1” + 3; // 13;
```

- Truthy vs Falsy is coercion in action

```
null; // false
“false”; // true
[]; // true
```

Refresher - What scope?

- What are the scopes here?

```
var a = 5;
function foo(b) {
   var c = 10;
   d = 15; // where is d?

   function bar(e) {
      var c = 2; // which c?
      a = 12; // which a?
   }
}
```

What scope, pt 2?

- What are the scopes here?

```
var a = 5;
function foo(b) {
  var c = 10;
  d = 15; // where is d?

  if (d < 5) {
    var c = 2; // which c?
  }
}
```

Exercise: Hoisting (pt 1 of 3)

- What will the output be?

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

 console.log(x); // what will the output be?
 return x;
}


foo();
```

Exercise: Hoisting (pt 1 of 3)

This…

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

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

Becomes…

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

 console.log(x); // 42
 return x;
}
foo();
```

Exercise: Hoisting (pt 2 of 3)

- And this?

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

Exercise: Hoisting (pt 2 of 3)

This…

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

Becomes…

```
function foo() {
 var x;
 console.log(x);// undefined
 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 bar;
function foo() {
 console.log("Foo!");
}


foo(); // Foo!
bar(); // TypeError


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

Exercise: Callbacks & Async

- What does this code do?

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

// what does this log out?
```

Solution: Callbacks & Async

```
for (var i = 1; i <= 5; i++) {
     (function(j){
        setTimeout(function() {
         console.log(i);
       }, i * 1000);
      })(i); // we use an IIFE to retain scope
} // outputs: 1, 2, 3, 4, 5
```

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: 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
```

Core JS concepts

- **All good?**
  - Data Types - *primitives and objects*
  - Coercion - *embrace it*
  - Scope - *function scop, it is lexical*
  - Hoisting - *it happens*
  - Object - *objects are everywhere*
  - Function declaration vs expression
  - Context - *it is dynamic*

If we're not OK on a topic here we *should* dive into it

WARM UP

Warm Up

- [just js] JavaScript Basics
  - http://jsfiddle.net/mrmorris/a5v1p5by/
- [dom + js] Input History
  - http://jsfiddle.net/mrmorris/t2wazjmg/

Solutions:
JavaScript Basics: http://jsfiddle.net/mrmorris/11u4vmkL/
Input History: http://jsfiddle.net/mrmorris/0hvt7d9e/

refresher

HTML

Wizard check

- OK with basic HTML?
- Can write a page in full?
- Write a **<form>** and all necessary input controls?
- Understand the difference between **<div>** and **<span>**?
- Understand the usage of **attributes** on elements
- When to use **id** versus **class**?

HTML

- **H**yper**T**ext **M**arkup **L**anguage
- Browsers allow support for all sorts of errors – html is very error tolerant
- Structure of the UI and "view data"
- Tree of element nodes
- HTML5
  - Rich feature set
  - Semantic
  - Cross-device compatibility
  - Easier!

Anatomy of a page

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        …document info and includes…
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Anatomy of an element

- <element attributeName="attributeValue">
  Content of element
  </element>
- Block vs inline
  - <p></p>
  - <strong></strong>
- Self closing elements
  - <input type="text" name="username" />

HTML Elements refresher

- **Structure**
  - <div>
  - <span>
  - <table>
    - <tr>, <td>, <thead>, <tbody>
  - <form>
    - <fieldset>, <label>, <input>, <select>, <textarea>
- **Content**
  - <h1> through <h6>
  - <p>
  - <ol> or <ul> (with<li>)
- **Text modifiers**
  - <em>, <strong>
- A list of elements:
  - https://developer.mozilla.org/en-US/docs/Web/HTML/Element

refresher

CSS

Wizard check

- OK with basic CSS selectors?
- Style a page in full?
- Select an element using CSS?
- Understand specificity?
- Got a few special pseudo-selectors under your belt?

Cascading Style Sheets

- Language for describing the look and formatting of the document
- Separates presentation from content

```
<!-- external resource -->
<link rel="stylesheet" type="text/css"
href="theme.css">


<!-- inline block -->

<style type=“text/css”>
     span {color: red;}
</style>


<!-- inline -->

<span style="color:red">RED</span>
```

Anatomy of a css declaration

-
  ```
  selectors {
      /* declaration block */
      property: value;
      property: value;
      property: val1 val2 val3 val4;
  }
  ```

-
  ```
  div {
      color: #f90;
      border: 1px solid #000;
      padding: 10px;
      margin: 5px 10px 3px 2px;
  }
  ```

CSS Selectors

- By element
  - h1 {color:#f90;} <h1></h1>
- By id
  - #header {} <div id="header"></div>
- By class
  - .main {} <div class="main"></div>
- By attribute
  - div[name="user"] {} <div name="user"></div>
- By relationship to other elements
  - li:nth-child(2) {} <ul><li></li>***<li></li>***</ul>
  - p span {} <p>**<span><span></span></span>**</p>
  - p > span {} <p>**<span><span></span></span>**</p>

CSS Specificity

- Selectors apply styles based on its **specificity**
  - inline, id, pseudo-classes, attributes, class, type, universal
- **`!important`** allows you to override

```
html:
<div id="main" class="fancy">
     What color will I be?
</div>

css:
#main{
   color: orange;
}
.fancy{
   color: blue;
}
#main.fancy{
   color: red;
}
```

refresher

THE DOM

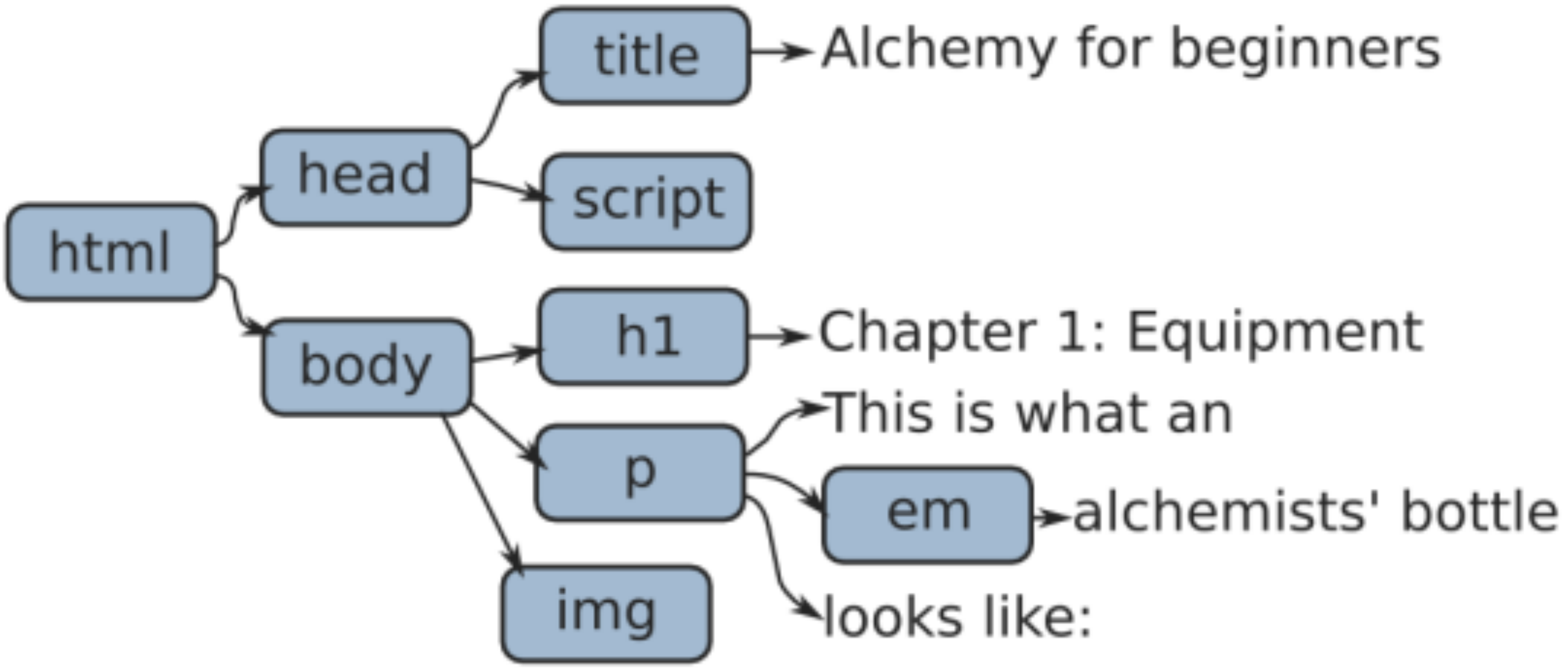
The DOM Refresher

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

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

Document Structure



Accessing elements

- Starting at **`document`** or a previously selected element
- `document.`**`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>
  ```
- **`.querySelectorAll`**`("p span");`
  ```
  // all elements that match the css selector
  // <p><span>Me!</span><span>Me!</span></p>
  ```

Element Traversal

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

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
- Or edit the element content directly
  - `elementVar.`**`innerHTML`**` = ‘<span>hi</span>’;`
  - `elementVar.`**`innerText`**` = ‘hi’;`

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`

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();
```

module

AJAX/XHR

AJAX/XHR

- Interface through which browsers can make HTTP Requests
- Handled by the `XMLHttpRequest` object
- Introduced by Microsoft in the 90s for ie, taken from there…
- Why use it?
  - Non-blocking
  - Dynamic page content/interaction
  - Supports many formats
- Limitations
  - Same-origin policy
  - History management

XHR– Step by step

1. Browser makes a request to a server
2. And the script continues along it's merry way

…some time later…

3. the server responds in xml/json/html
4. Browser parses and processes response
5. Browser invokes our JavaScript callback

Making the request

- Create a request object, begin the request, define headers, send it

```
var req = new XMLHttpRequest();

// attach listener (next slide)

req.open(“GET”, “url.json”, true);
// set header after open but before send
// defaults to Accept */*
req.setRequestHeader("Accept","application/json");
req.send(null);
```

Handle the response

- “load” event will fire when response is received
- Request object will have `responseText` and `status`

```
req.addEventListener(“load”, function(e) {
      // HTTP status codes
      if (req.status == 200) {
            console.log(req.responseText);
      }
});
```

XHR Example

- Loading content from a weather API
  - http://jsfiddle.net/mrmorris/cfwa8v92/

Data formats

Format	Summary	PROS	CONS
HTML	Easiest for content in page	• Easy to parse • No need to process much	• Server must produce the HTML • Data portability is limited • Limited to same domain
XML	Looks similar to HTML, more strict	• Flexible and can handle complex structure • Processed using the DOM	• Very verbose, lots of data • Lots of code needed to process result • Same domain only
JSON	Similar object literal syntax	• concise! Small • Easy to use within JavaScript • Any domain, w/ JSONP or CORS	• Syntax is strict • Can contain malicious content since it can be parsed as JavaScript

XHR with HTML

- Easiest way to go
- Works with the DOM and styles
- Scripts will NOT run

XHR with XML

- More work in processing the data to turn XML into HTML

```
var data = xhr.responseXML;
var events = data.getElementsByTagName('event');

for (var i=0; i<events.length; i++) {
  var container = document.createElement('div');
  container.className = 'event';
  // create img node
  // append
}
```

JSON

- **J**ava**S**cript **O**bject **N**otation
- Most commonly used web data communication format
- Like an object literal, except:
  - Property names must be surrounded by double quotes
  - No function definitions, function calls or variables
- Methods
  - `JSON.stringify(object);`
  - `JSON.parse(string);`

JSON

```
{
  name: “Jason”,
  trophies: [
    “trophy1”,
    “trophy2”
  ],
  sayHi: function() {
    console.log(‘hi’);
  }
  age: user.age,
  car: {
    name: “toyota”,
    year: 1985
  }
}
```

to json

```
{
  “name”: “Jason”,
  “trophies”: [
     “trophy1”,
     “trophy2”
  ],
  “age”: 40,
  “car”: {
     “name”: “toyota”,
     “year”: 1985
  }
}
```

XHR with JSON

- It is sent and received as a string and will need to be de-serialized

```
var data = JSON.parse(xhr.responseText);
var newContent = “”;

for (var i=0; i< data.length; i++) {
  newContent += ‘<div class=“event”>’;
  newContent += ‘<img src=“’ + data[i].val+ ‘”/>’;
}

document
 .getElementById(‘content’)
 .innerHTML = newContent;
```

XHR usage

- It is best to use an abstraction of `XMLHttpRequest` for
  - `status` and `statusCode` handling
  - Error handling
  - Callback registration (`onreadystatechange` vs `onload`)
  - Browser variations and fallbacks
  - Additional event handling
    - `progress`
    - `load`
    - `error`
    - `abort`
- Use a lib like jQuery….

Cross-origin

- By default, ajax requests must be made on the same domain
- Alternatives to this are:
  - A proxy file on the server
  - JSON/p “Json with padding”
  - CORS (Cross-origin resource sharing), which involves new http headers between browser and server – ie10+
- For later: http://jsonplaceholder.typicode.com/

JSONP

- Browsers don't enforce same-origin policy on the `src` in script tags
- Shenanigans:
  - We define a handling **callback** function
  - We dynamically add a **script** referencing an external `script`
  - We tell the script the **callback** to wrap the response in
  - Once script loads, the response is wrapped in our **callback**, which is invoked on load
- Caveats
  - Only works with GET requests
  - Does **not** use XMLHttpRequest
  - Super insecure and shouldn't ever be used in conjunction with untrusted third parties due to CSRF, XSS and other exploits

CORS

- Cross-Origin Resource Sharing
- A set of headers sent by the requesting client (XHR) and the responding server that can negotiate whom can request what from where
- Caveats
  - Supports **all** HTTP verbs
  - Usable with XMLHttpRequest
  - Simple in theory, complex in practice

XHR Recap

- A means for the browser to make additional requests without reloading the page
- Enables very fast and dynamic web pages
- Best with small, light transactions
- JSON is the data format of choice
- Requests across domains are possible but require jumping through some extra hoops (and your server must support it)

Exercise - XHR

- **Pure JS XHR**
  Load content from a CORS-ready endpoint
  - Let’s check out our API:
    - https://jsonplaceholder.typicode.com/posts
  - Fork me:
    - https://jsfiddle.net/mrmorris/nwq9kaxn/

Solutions:
Pure JS XHR: https://jsfiddle.net/mrmorris/q13yckz6/

module

JQUERY AJAX

jQuery Ajax

- Several ways to do this, but they are all shortcuts of **$.ajax()**

```
var jxhr = $.ajax({
    type: string (GET or POST)
    url: string
    data: mixed (converted to query str)
    success: function
    error: function
    complete: function
    timeout: number
    dataType: string (xml, json or html)
    beforeSend: function
}); // returns jQuery xhr object
```

jQuery – handling the response

- Callbacks (deprecated in jQuery 3.0)
  - beforeSend, dataFilter, success, error, complete
- or…jQuery XHR implements Promise interface

  -
    ```
    var prom = $.ajax({…});

    prom.done(function(response){…});
    prom.fail(function(){…});
    prom.always(function(){…});
    prom.then(doneFn, failFn);
    ```
- These promise methods can be chained
  - `$.ajax().done().fail().always()`

jQuery ajax shorthand

- element.load()
  - Loads data directly into an element
  - Can target fragment elements in the response
    - $(‘#content’).load(‘bla.html#content’);
- $.get(settings);
- $.post(settings);
- $.getJSON(settings);
- $.getScript(settings);

jQuery Ajax examples

- More $.ajax with form post
  - http://jsfiddle.net/mrmorris/pj4e7jxv/
- Example with CORS API
  - http://jsfiddle.net/mrmorris/5vwcx7zp/

Promises

- Ajax requests are returning a promise
  - Actually a “jqXHR” object that implements the *Promise interface*
- Promises have a lifecycle
  - Unfulfilled
  - Fulfilled
  - Failed
- In jQuery, the Promise is based off the $.Deferred object

The advantages of a promise

- You can:
  - add multiple success/failure callbacks
  - add callbacks even after the Promise lifecycle is complete
- Use the behavior of Deferred objects
  - Like delay a callback until multiple promises are complete
  - Or pipe result data
- The result of an asynchronous operation(s) can be treated as a first class object
- A solution to “callback hell”
  - Think of it like async pathways

jQuery Ajax review

-
  ```
  $.ajax({
      type: 'GET', // or 'POST', 'DELETE',
      data: {},
      ~~success~~: callback
      ~~error~~: callback
      ~~complete~~: callback
      dataType: 'json', // 'json','html'
  });
  ```
- $.ajax (and shortcuts) method immediately (synchronously) returns a Promise object

  -
    ```
    var prom = $.ajax({…});
    prom.done(function(response){…});
    prom.fail(function(){…});
    prom.always(function(){…});
    ```
- These promise methods can be chained
  - `prom.done().fail().always()`

Anatomy of an Ajax request

```
var prom = $.ajax({
        type: 'GET',
        url: 'http://some.api.com/data.json',
        dataType: 'json',
        data: {}
});
prom.done(function(response, status, prom) {
        // process your response data
});
prom.fail(function(prom, status, error) {
        // handle the error
});
prom.always(function(response, status, error) {
        // wrap up after done or fail
});
// combined done/error
prom.then(doneCallback, failCallback);
```

Exercise – jQuery Ajax

- We'll be using this API:
  - http://jsonplaceholder.typicode.com/
  - https://github.com/typicode/jsonplaceholder
- **Photo Grid**
  Working together lets complete a dynamic photo grid
  - http://jsfiddle.net/mrmorris/Ln8ecynw/
  - Hint: Check out the network panel
- **Todo List**
  Use ajax to load content from an API to build up a todo list
  - http://jsfiddle.net/mrmorris/1gtqsohv/

Solutions:
Photo Grid: http://jsfiddle.net/mrmorris/x38yzpc2/

module

OBJECTS

Objects

- Remember that everything is an object except **`null`** and **`undefined`**
  - Even primitive literals (numbers, strings, etc) have object wrappers
- An object is a dynamic collection of properties
- `this` refers to the object a function is invoked on

```
var dog = {
 name: ‘Fido’,
 age: 10
}
dog.speak = function() {
  console.log(this.name, “says”, “Bark!”);
}
dog.speak(); // Fido says Bark!
```

Object Creation in JavaScript

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

The Object Literal

- Create an object literal with {}:

```
var myObjLiteral = {
   name: “Mr Object”,
   age: 99,
   toString = function() {
      return this.name; // this?
   }
};
```

- http://jsfiddle.net/mrmorris/4dsLonat/

Object properties

- Can get/set with dot or array-access syntax

```
myObj.key;
myObj.key = 5;

myObj[“key”];
myObj[“key”] = 5;

var propName = “key”;
myObj[propName] = 5;
```

- Can delete a property with `delete`

```
delete myObj.key;
```

Object reflection

- Objects **inherit** properties from their prototype
  - ex: Array inherits from Object
  - "**Own**" means the property exists on the object itself, not from up the **prototype chain**
  - Use **`in`** and **`hasOwnProperty`** to determine where property resides

```
var myObj = { name: 'Jim' };
myObj.toString(); // [object Object]

'name' in myObj; // true!
'toString' in myObj; // true
myObj.hasOwnProperty('toString'); // false!
```

Object reflection, continued

- `Object.keys(obj)`
  - Returns array of all “**own**”, enumerable properties
- `Object.getOwnPropertyNames(obj)`
  - Returns array of all “**own**“ property names, including non-enumerable

Enumerating over objects

- `for…in`
  - Over object properties
- `for…of (ES6)`
  - Over *iterable* values
- ~~`for each…in`~~
  - deprecated
  - over object properties

for…in

- Loop over ***enumerable properties*** of an object
  - Will include inherited properties as well, including stuff you probably don't want
    - Use `obj.hasOwnProperty(propertyName)`
  - In order of insertion of the property

```
var obj = {foo: true, bar: false};


for (var prop in obj) {
  if (obj.hasOwnProperty(prop)){
    console.log(prop);
  }
  obj[prop];     // true
} // outputs: foo, bar
```

for...of [ES6]

- Loop over ***enumerable values*** of an **iterable**
  - Will include inherited properties as well, including stuff you probably don't want
  - **Not just objects** — *iterables* (including arrays)

```
var obj = {foo: true, bar: false};


for (let val of iterableThing) {
  console.log(val);
} // true, false


for (let x of [1,2,3]) {
  console.log(x);
} // 1, 2, 3
```

Mutability

- All primitives in JavaScript are immutable
  - Using an assignment operator just creates a new instance of the primitive
  - Pass-by-value
  - Unless you used an object constructor for a primitive…
- Objects are mutable (and pass-by-reference)
  - Their values (properties) can change

Exercise - Mutations

- What will the result of this be:

```
var rabbit = {name: ‘Tim’};
var hp = 100;


function attack(obj, hp) {
  obj.fight = true;
  hp = 10;
}


attack(rabbit);
console.log(hp, rabbit); // ???
```

Exercise - Mutations

- What will the result of this be:

```
var rabbit = {name: ‘Tim’};
var hp = 100;


function attack(obj, hp) {
  obj.fight = true;
  hp = 10;
}


attack(rabbit);
console.log(hp, rabbit); // ???
```

Properties descriptors

- Object properties have **descriptors**
- They modify property behavior

```
var myObj = {};
Object.defineProperty(myObj, “key”, {
    value: 5,
    enumerable: true, // included in loop
    configurable: false, // re-configurable
    writable: false, // re-assignable
}
myObj.key = 10; // silently fails
```

Object getter/setter

```
var myObj = {
  log: [‘test’],
  get latest() {
    if (this.log.length) {
      return this.log[this.log.length-1];
    }
    return undefined;
  }
}
Object.defineProperty(myObj, "newProp", {
    set: function(value) {
        this.bla = value;
    }
});
```

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);
```

Exercise - Objects

- **Objectify Yourself**
  - Fork: https://jsfiddle.net/mrmorris/rt5z9mo0/

Solutions:
Objectify Yourself - https://jsfiddle.net/mrmorris/d2847z01/

module

BUILT-IN OBJECTS

Built-in Objects

- String
- Number
- Boolean
- Function
- Array
- Date
- Math
- RegExp
- Error
- http://jsfiddle.net/mrmorris/rrb67ev0/

String

- Instance properties

```
new String('foo').length // 3
```

- Instance method examples

```
var str = new String('hello');
str.charAt(0);      // 'h'
str.concat('!');    // ‘hello!'
str.indexOf('w'); // 6
str.slice(0, 5);   // 'hello'
str.substr(6, 5); // 'world'
str.toUpperCase(); // 'HELLO!'
```

Number

- Generics

```
Number.MIN_VALUE
Number.MAX_VALUE
Number.NaN
Number.POSITIVE_INFINITY
Number.NEGATIVE_INFINTY
```

- Instance method examples

```
var num = new Number(3.1415);
num.toExponential();       // "3.1415e+0"
num.toFixed();             // 3
num.toPrecision(3);        // 3.14
```

Number Properties

- Properties
  - `MAX_VALUE`
  - `NaN`
  - Etc…
- Generic methods
  - Number.isInteger()
  - Number.isFinite()
  - Number.parseFloat()
  - Number.parseInt()
- Instance methods
  - `num.toString()`
  - `num.toFixed()`
  - `num.toExponential()`

Math

- Singleton-ish
- Methods
  - `abs, log, max, min, pow, sqrt, sin, floor, ceil, random…`
- Properties
  - `E, LN2, LOG2E, PI, SQRT2…`

Array

- Generics

  Array.**isArray**([])   // true

- Examples
  - http://jsfiddle.net/jmcneese/qsxgvdnn

Array mutator methods

```
var arr = new Array(1, 2, 3);
arr.pop();                    // 3
arr.push(3);                  // 3
arr.reverse();                    // [3, 2, 1]
arr.shift();                  // 3
arr.sort();                   // [1, 2]
arr.splice(1, 0, 1.5);        // [1, 1.5,
2]
arr.unshift(0);               // [0, 1, 1.5,
2]
```

Array accessor methods

```
var arr = new Array(1, 1);
arr.concat([2, 4]);   // [1, 1, 2, 4]
arr.join('-');            // "1-1"
arr.slice(1, 1);      // [1]
arr.toString();       // "1,1"
arr.indexOf(2);       // -1
arr.lastIndexOf(1);   // 1
```

Array iteration methods

```
var arr = new Array(1, 1, 2, 4);
arr.forEach(fn);
arr.every(fn);
arr.some(fn);
arr.filter(fn); // new array filtered
arr.map(fn); // new array transformed
arr.reduce(fn); // result from an
array
arr.reduceRight(fn);
```

Array enumeration

- Use “`for`” not “`for…in`”, which doesn’t keep array keys in order

  -
    ```
    for (var i=0; i < myArray.length; i++) {

    }
    ```
- `.forEach(callback[, thisArg])`
  - New in es5 (ie9+)
  - No way to stop or break a `forEach` loop

  -
    ```
    myArray.forEach(function(val, index, arr)
    {

    });
    ```

Array tests

- `arr.every(callback[, thisArg])`
  - checks If every element in an array passes a callback function

  -
    ```
    myArray.every(function(val, index, arr) {
     return (val>0); // evaluates to boolean
    });
    ```
- `arr.some(callback[, thisArg])`
  - verify if at least one passes the test

Array filter, map

- `.filter()`
  - Iterate over your array of items passing them to a function. Returning `true` from the function indicates the item should be retained.

    -
      ```
      myArray.filter(function(item) {
            return item!=2;
      }); // removes items that don't equal 2
      ```
- `.map()`
  - Iterates over array, invoking a function on each value. The return value is the modified value of the item.
- http://jsfiddle.net/mrmorris/pbwy3hy5/

Array reduce()

- `.reduce()`
  - Boils down a list of values into a single value.

```
[0,1,2,3,4].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);
// initial acc value can be passed in
```

Date

- Represents a single moment in time based on the number of milliseconds since 1 January, 1970 UTC

```
    new Date();
    new Date(value);
    new Date(dateString);
    new Date(year, month[, day[, 
hour[, minutes[, seconds[, 
milliseconds]]]]]);
```

- Examples
  - http://jsfiddle.net/jmcneese/76aat2kc

Date Methods

- Generics

```
Date.now()
Date.parse('2015-01-01')
Date.UTC(2015, 0, 1)
```

- Instance method examples

```
var d = new Date();
d.getFullYear();        // 2015
d.getMonth();              // 7
d.getDate();            // 15
```

RegExp

- Creates a regular expression object for matching text with a pattern

```
var re = new RegExp("\w+", "g");
var re = /\w+/g;
```

- Generics

```
var re = new RegExp("\w+", "g");
```

  re.**global**;   // true
  re.**ignoreCase**;   // false
  re.**multiline**; // false
  re.**source**;   // "\w+"

- Examples
  - http://jsfiddle.net/jmcneese/8jnso5wf

RegExp Methods

- Instance methods
  - re.**exec**(*str*)
  - re.**test**(*str*)
- String methods that accept RegExp params
  - str.**match**(*regexp*); // array of matches
  - str.**replace**(*regexp, replacement*); // string with replacement
  - str.**search**(*regexp*); // returns 1 at first match
  - str.**split**(*regexp, limit*); // returns array

Regular Expressions matchers

- Escape with / backslash
- Use [] for sets, [01234] or [0-4] for a range
- Special character groups, ex: \d (digits) and \w (alphanumeric)
- + match at least one
- * match 0 or more
- ^ invert
- ? Optional - /neighbou?r/
- {4} time occurrence
- {2,4} – at least twice, at most 4 times

Error

- Error objects are thrown when runtime errors occur
  - Can also be used as a base objects for user-defined exceptions

```
var err = new Error('Oh noes!');
```

- Implementation varies across vendors
- Instance properties

```
err.name;          // "Error"
err.message;       // "Oh noes!"
```

Error Handling

- JavaScript is very lenient when it comes to handling errors
- Internal errors are raised via the **throw** keyword, and are then considered "exceptions"
- Exceptions are handled via a **try/catch/finally** construct, where the thrown exception is passed to the **catch** block
  - Nesting allowed
  - Exceptions can be re-thrown
- *Anything* can be thrown, of any data type
- Uncaught exceptions halt the overall script
- Example
  - http://jsfiddle.net/jmcneese/m83pgvbn

Built-in Errors

- Error (Top level object)
- SyntaxError
- ReferenceError
- TypeError
- RangeError
- URIError
- EvalError

Exercise – Core objects

- **Data Grids**
  Display an array of objects as a table in the console
  - http://jsfiddle.net/mrmorris/0kptbv7p/
- **Arrays**
  Filtering and mutating arrays
  - https://jsfiddle.net/mrmorris/ce7s09j0/
- **Strings**
  Replacing a word in a string
  - https://jsfiddle.net/mrmorris/owrtzequ/

Solutions:
Data Grid http://jsfiddle.net/mrmorris/5kfLhn8a/
Arrays https://jsfiddle.net/mrmorris/bptg1mkw/
Strings https://jsfiddle.net/mrmorris/oc2ba3jj/

module

FUNCTIONS

Functions: "The best part of JS"

- Reusable, callable blocks of code
- Functions can be used as:
  - Object methods
  - Object constructors
  - Modules and namespaces
- They *are* ***First Class Objects***
  - *Can have their own properties and methods*
  - *Can be passed as function arguments (higher order!)*
  - *Can be referenced by variables*

Function Usage

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

Defining a function

- Four ways
  - Function **declaration**
  - Function **expression**
  - `Function()` **constructor**
  - **Fat arrow** [ES6+]
- A bunch of examples:
  - http://jsfiddle.net/mrmorris/N8vcg/

Function Declaration

```
// declaration
function adder(a, b) {
     return a + b;
}


// invokation
adder(1, 2); // 3
```

- The function name is *mandatory*
- Function declarations are ***hoisted*** to the top of the scope; available for entire scope

Function Expressions

```
// function expression
var adder = function(a, b) {
      return a + b;
}


// invokation is identical
adder(1, 2); // 3
```

- Define a function and assigns it to a variable
- Function name is optional — *making it anonymous*

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 your anonymous functions
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 sumAll() {
  // 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);

  return args.reduce(function(acc, curr) {
     return acc + curr;
  });
}
sumAll(1, 2, 3); // ?
```

Functions as First Class Objects

```
// function passed in to another function
setTimeout(function() {
  console.log('HI!');
}, 1000);


// check the docs; we define argument names
[1,2,3].forEach(function(curr, i, arr) {
  console.log(curr, i, arr);
});
```

- Functions can be passed around as arguments
- We can define argument names when we define per an api/interface

Default Values [ES6]

- ES6

```
function adder(first, second = 1) {
   // body
}
function addComment(comment = getComment()) {
   // body
}
```

- Pre-ES6

```
function adder(first, second) {
   second = second || 1;
}
```

Scope & Context

- Functions have **scope**
  - Determines visibility of variables
  - Lexical scope (write-time)
- There is also ***Context***
  - *R*efers to the location a function/method was invoked *from*
  - Like a *dynamic scope*; it is defined at run-time
  - Context is referenced by a keyword in all functions: `this`

this?

- Anyone have an idea what **this** is?

```
function runMe() {
   console.log(this);
}

runMe(); // ?
```

`this` is context

- Reference to an object
  - The ***context*** where the function is running
  - *“The object of my invokation”* 🌹
- Dynamically bound
  - Determined on invokation
  - Not lexical
- Basis of
  - Inheritance
  - Multi-purpose functions
  - Method awareness of their objects

this example

```
var person = {
  name: “Carol Danvers”,
  speak: function() {
    console.log(“Hi, I am”, 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(); // ?
```

Explicit binding

- Context can be changed via a Function’s `call`, `apply` and `bind` methods

```
obj.foo(); // obj context
obj.foo.call(window); // window context
```

- “`bind`” returns a copy of the function with the context re-defined.

```
var getX = module.getX;
boundGetX = getX.bind(module);
```

- http://jsfiddle.net/mrmorris/or7y5orn/

Example: Explicit binding

```
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);
```

Example: Binding 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
```

Arrow Functions [ES6]

- (**Fat) Arrow** functions
  - Super short function syntax
  - Always anonymous
  - Lexical contextual binding
- Caveats
  - No **`arguments`** of its own (the *outer* function's args)
  - No **`this`** of its own (uses the enclosing context)

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


// can instead be written as
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?
}

me = {
  name: ‘Tim’,
  talk: (x) => {
    console.log(this.name, x); // this is global :(
  },
  talkLater: function () {
    setTimeout(() => {console.log(this.name)}, 1000); // this is me :D
  }
}
```

“The same `this` inside the function as outside the function”.

Bound on creation (not invokation)

module

CONTEXT

Scope & Context

- We already discussed ***Scope***
  - Determines visibility of variables
  - Lexical scope (write-time)
- There is also ***Context***
  - *R*efers to the location a function/method was invoked *from*
  - Like a *dynamic scope*; it is defined at run-time
  - Context is referenced by a keyword in all functions: `this`

this?

- Anyone have an idea what **this** is?

```
function runMe() {
   console.log(this);
}

runMe(); // ?
```

`this` is context

- Reference to an object
  - The ***context*** where the function is running
  - *“The object of my invokation”* 🌹
- Dynamically bound
  - Determined on invokation
  - Not lexical
- Basis of
  - Inheritance
  - Multi-purpose functions
  - Method awareness of their objects

this example

```
var person = {
  name: “Carol Danvers”,
  speak: function() {
    console.log(“Hi, I am”, 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(); // ?
```

Binding context

- Default binding
  - Global
- Implicit binding
  - Object method
    - Warning: Inside an inner function of an object method it refers to the global object
- Explicit binding
  - Set with `.call()` or `.apply()`
- Hard binding
  - Set with `.bind()`
- Constructorbinding with "`new`" keyword
- http://jsfiddle.net/mrmorris/RUNS5/

“this” and global

- It’s possible to “leak” and access the global object when invoking functions that reference this from outside objects

  -
    ```
    var setName = function(name) {
       this.name = name;
    }
    setName(‘Tim’);
    name; // “Tim”
    window.name === name; // true! oops.
    ```

- “use strict” prevents leaks like that by keeping global “this” undefined in this case

Explicit binding

- Context can be changed via a Function's `call`, `apply` and `bind` methods

```
obj.foo(); // obj context
obj.foo.call(window); // window context
```

- "`bind`" returns a copy of the function with the context re-defined.

```
var getX = module.getX;
boundGetX = getX.bind(module);
```

- http://jsfiddle.net/mrmorris/or7y5orn/

Example: Explicit binding

```
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);
```

Example: Binding 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(); // ?
```

Arrow Functions [ES6]

- (**Fat) Arrow** functions
  - Super short function syntax
  - Always anonymous
  - Lexical contextual binding
- Caveats
  - No **`arguments`** of its own (the *outer* function's args)
  - No **`this`** of its own (uses the enclosing context)

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


// can instead be written as
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?
}


me = {
  name: ‘Tim’,
  talk: (x) => {
    console.log(this.name, x); // this is global :(
  },
  talkLater: function () {
    setTimeout(() => {console.log(this.name)}, 1000); // this is me :D
  }
}
```

“The same `this` inside the function as outside the function”.

Bound on creation (not invokation)

Exercise - Objects

- **Objectify Yourself**
  - Fork: https://jsfiddle.net/mrmorris/rt5z9mo0/

Solutions:
Objectify Yourself - https://jsfiddle.net/mrmorris/d2847z01/

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
```

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

Exercise - IFFE and Closures

- **Hosts Module**
  - https://jsfiddle.net/mrmorris/gv3ns9m5/
- **Private Collection**
  - In your Objectify Me lab, go back and make "trophies" a private variable with a getTrophy(i) accessor.

Solutions:
Hosts Module - https://jsfiddle.net/mrmorris/1z1sb5so/
Objectify Yourself private - https://jsfiddle.net/mrmorris/wocw3b1v/

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**
  - `var me = {name: ‘Tim’};`
- **`Object.create(personObj)`**
  - `var me = Object.create(null);`
- **Constructors w/ `new`**
  - `var me = new Person(‘Tim’);`
- **Factory Functions**
  - `var me = makePerson({name: ‘Tim’});`
- **ES6 `class` keyword**
  - `var me = new Person(‘Time’);`

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
  - …

  - *What’s something we want to work with?*
    - Animals
    - Vehicles
    - Washing Machines?

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!”);
  }
}
```

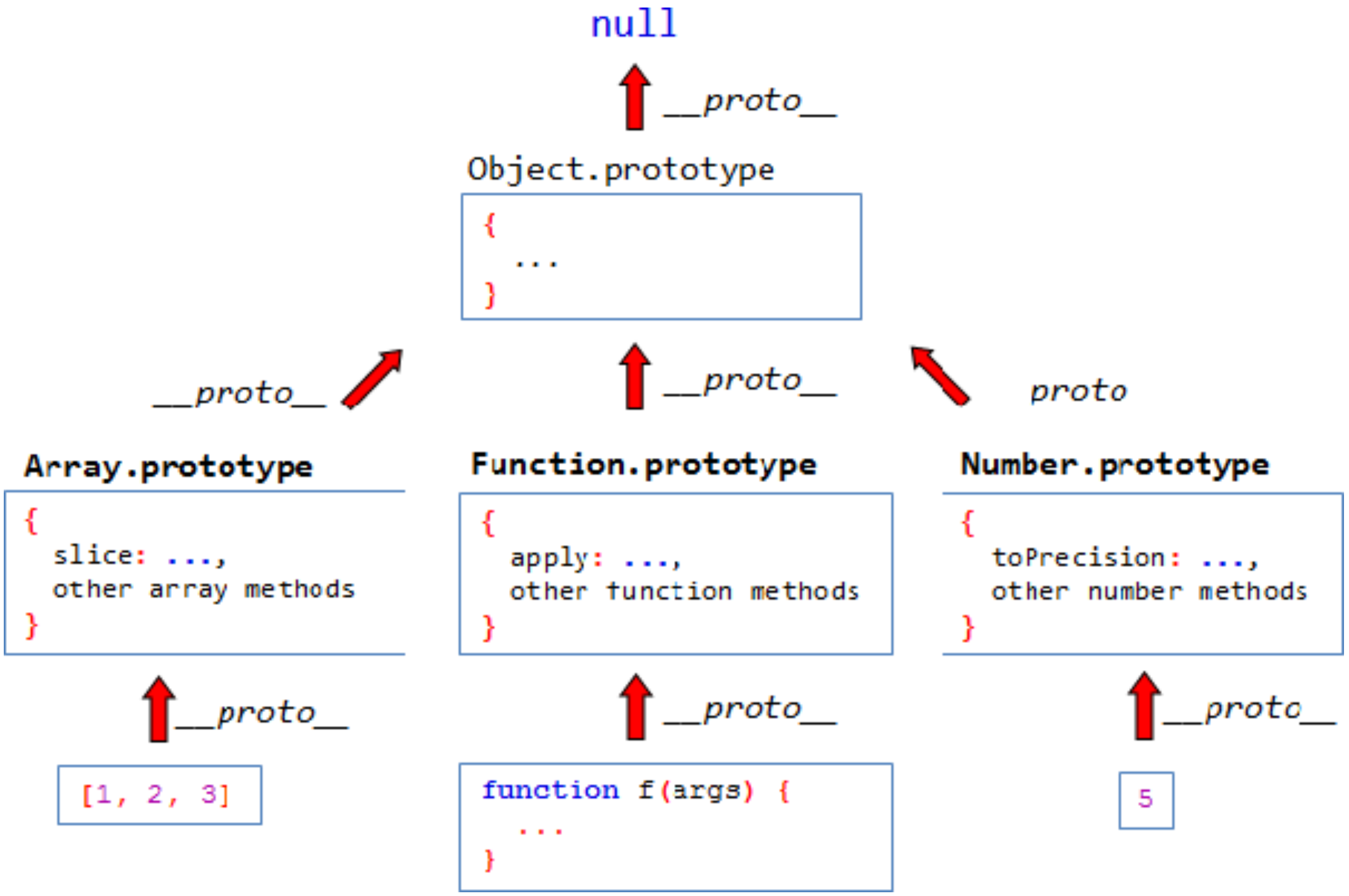
Prototypal Inheritance

```
// 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



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 Methods

- **Setting the Prototype**
  - `obj.__proto__ = proto; //slow`
  - `var obj = Object.create(proto); // fav`

  -
    ```
    MyFunction.prototype = proto;
    var obj = new MyFunction(); // class-like
    ```
  - `Object.setPrototypeOf(obj, proto); // slow`

- **Reading the prototype**
  - `Object.getPrototypeOf(obj);`
  - `obj.__proto__;`

- **Set without prototype?**
  - `var obj = Object.create(null); // “plain object”`

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 and 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 new/constructors do

```
var MyConstructor = function(name) {
  this = {}

  // set instance-level properties
  this.name = name;

  this.__proto__ = MyConstructor.prototype;
  return this;
}
```

- What it does exactly…
  1. Uses **`this`** to set own properties on a new object
  2. Set's the **`[[prototype]]`** 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 continued

```
// we want: dog -> Dog.prototype -> Animal.prototype

// our superclass
var Animal = function (sound) {
  this.sound = sound;
}
Animal.protootoype = {/*… 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;
```

Exercise - What's wrong here?

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.walk = function() {
  alert(this.name + ' walks');
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = Animal.prototype;

Rabbit.prototype.walk = function() {
  alert(this.name + " bounces!");
};
```

Factory Function Pattern

- Functions that create and return objects
- Alternative to constructors
- Better encapsulation & privacy
- 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);

var dogCat = Barker(cat);
```

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
```

`Class` keyword [ES6]

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

Without class

```
var Human = function(name) {
  this.name = name;
}

Human.prototype.talk =
function(str) {
  console.log(this.name, "says",
str);
}

let tim = new Human('tim');
tim.talk('Hi!');
```

With class

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

  talk(str) {
    console.log(this.name,
"says", str);
  }
}

let tim = new Human('tim');
tim.talk('Hi!');
```

Extending Classes

```
var Rectangle = class {
  constructor(width, height) {
    this.width = width;
    this.height = height;
  }
  // no literal properties allowed
  get area() {
    return this.width * this.height;
  }
}

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

`Class` keyword extras

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

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!

OO – Exercise

- **Create a hierarchy of objects**
  - Cats, Dogs, Animals
  - Me, People, Mammals
  - Car, Truck, Vehicles
- **Collections and Items**

  Create a Collection object that contains a set of Item objects.
  - Fork me:
    - http://jsfiddle.net/mrmorris/kobseonk/

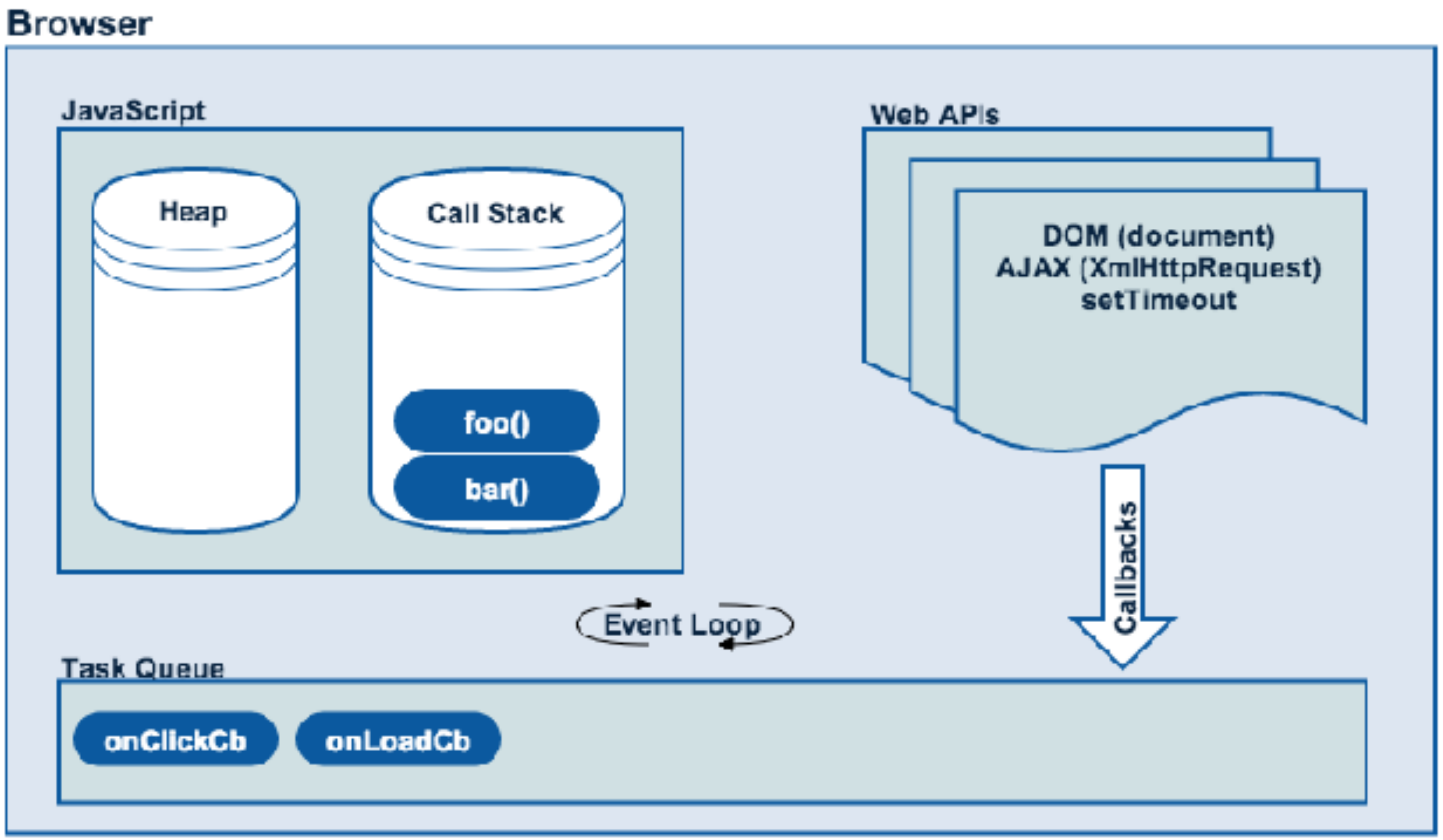
Solutions:
Collections and Items (no bonus): http://jsfiddle.net/mrmorris/3acj3f4r/

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`
  - Observables

Callback Pattern

- A function passed to another function as a parameter
  - …so that it can be invoked later by the calling function.
- Aren’t asynchronous on their own
  - …but we tend to use them for such things
  - ex: event handling, ajax handling, file operations, etc

```
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) {
                      /*…*/
                });
          });
    });
});
```

Promises

- A **Promise** represents a proxy for a value not necessarily known when the promise is created
  - They represent the *promise of future value*
- **Benefits:**
  - Guarantees that callbacks are invoked
  - Composable (can be chained)
  - Immutable (one-way latch)
  - You can continue to use them after resolved
  - They are objects you can pass around
- **Bummers:**
  - ES6+
  - No `.finally()`

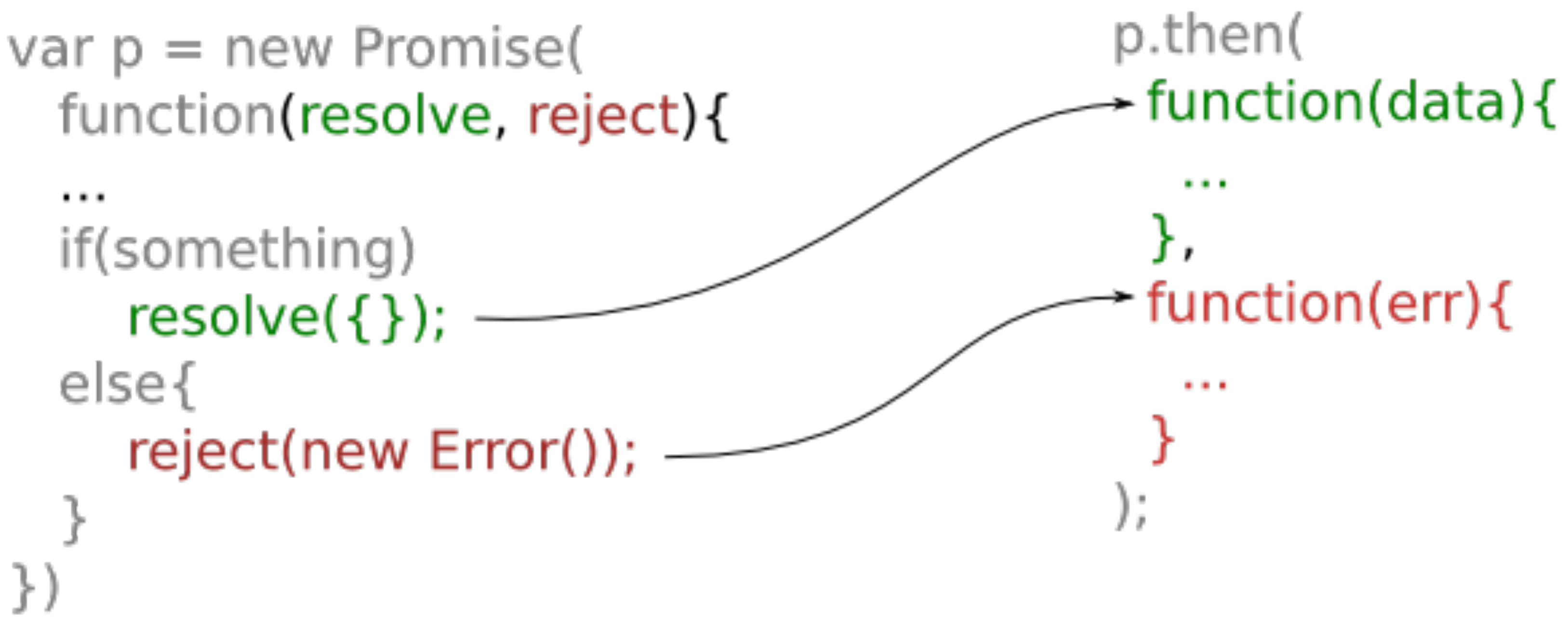
Making a Promise

- 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

- `fulfill()` and `reject()` don't explicitly return from the constructor
- Handle errors thrown
  - Use the reject/error handler argument in `then()`
  - 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 any return value as a new Promise
    - …can chain them
    - you can specify a *new* promise to return
    - in this way you can have a waterfall of operations dependent on the previous completing

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

promise1.then(function(data){
    console.log(data); // 5
    return data + 2; // returns a new promise
}).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 [ES6]

- Two new keywords allow us to write asynchronous code that looks and feels synchronous
- **`async function`**
  - Defines an asynchronous Function that can **yield flow of control** back to the caller
  - The function immediately returns a Promise that will be resolved when the function returns a value or rejected when it has an error
    - The function is resolved with any return value
    - Errors with any error thrown
- **`await`**
  - Informs code *within* an async function to yield/wait for an *internal* Promise to resolve before proceeding

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);
}

var rendered = getAndRenderArtists();
rendered.then(function(response) {
   console.log(‘Page is loaded’);
});
```

Exercise - Promises

- **callLater**
  A function that sets up a waterfall of promises
  - Fork: https://jsfiddle.net/mrmorris/kp4gqp69/
- **Ajax with Promises**
  Set up an Ajax utility object that makes ajax requests and returns a promise
  - Fork: https://jsfiddle.net/mrmorris/5yzby96w/

Solutions:
callLater - https://jsfiddle.net/mrmorris/sLbmmq4g/
Ajax with Promises - https://jsfiddle.net/mrmorris/oa1jbgr3/

module

OBSERVABLES

Observer pattern

- When an object (the subject or observable) maintains a list of subscribers (observers) and notifies them of any state changes.

-wikipedia

Observer pattern

- **When**
  - something happens
  - data changes
  - data is provided
  - an event occurs
- **Then**
  - trigger functionality
  - call a function
  - let something else know
  - update the world

Observers in the browser

- Events!
  - we subscribe with addEventListener
  - event is “pushed” to our handler
  - we unsubscribe with removeEventListener
- Data-binding
  - When an object changes
  - Update the view
  - And vice-versa (two-way)
- Generic observables (*meta-code)…
  - mousemove = Observable.from(‘mousemove’);
  - mousemove.subscribe(handler);
  - mousemove.unsubscribe();

The Observable Object

- An implementation of the **observer pattern**
  - Or pub-sub
- **Generalizing a collection that arrives over time**
  - Something has data over time
  - We can subscribe to it
  - Trigger functionality as each piece of data arrives
    - Also handle when it's done (if ever)
    - Or has an error
- Why? **Adapt all our async apis** into one api
  - dom events, websockets, sse, streams, service workers, xhr, setInterval

Promises and streams

- *"Observable (sorta) like a stream of promises"*
- Promises are great, but…
  - They are just syntactic sugar on top of callbacks
  - Act on data and return
  - Are fulfilled once
  - Can't handle a stream of data, or process that returns data over time but is never fulfilled

What it looks like

- An object determines what is observable
  - A data stream (api?)
  - Events (mousemove?)
  - An array?
- The observable is responsible for broadly informing of events
  - next, complete, error
- Then anything can subscribe to the Observable as long as they follow the interface
  - next, complete, error

What it looks like

```
var resize = new Observable((o) => {
  // listen for window resize
  window.addEventListener("resize", () => {
    var height = window.innerHeight;
    var width = window.innerWidth;
    o.next({height, width});
  });
  return () => {
    // function that removes listener
  }
});

var subscribed = resize.observe({
  next: (value) => {
    console.log(“Value is:”, value);
  }
});

subscribed.unsubscribe();
```

Observable support

- ECMAScript proposal in the works
  - https://github.com/tc39/proposal-observable
- Reactive JS (RxJS)
  - http://reactivex.io/
- Bacon
  - https://baconjs.github.io/

the end is hear

WRAPPING UP

Going beyond

- Modules
- jQuery toolkits
  - Help with modules
  - Minify and compile
  - Transpile
- HTML5 Apis
  - Web Workers
  - Sockets
- JS in the server
  - NodeJS

Stay sharp

- Solve small challenges for kata
  - http://www.codewars.com/
- Code interactively
  - http://www.codecademy.com/
- Share your code and get feedback
  - http://jsfiddle.net
- Free e-book
  - http://eloquentjavascript.net/
- Re-introduction to JavaScript
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Go now and code well

- That's a wrap!
  - What did you enjoy learning about the most?
  - What is your key takeaway?
  - What do you wish we did differently?
- Any other comments, questions, suggestions?
- Feel free to contact me at mr.morris@gmail.com or my eerily silent twitter **@mrmorris**