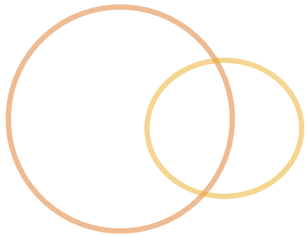
Advanced JavaScript

Joshua McNeese
Develop Intelligence

Introductions



- Who am I?
- Who are you?
- What do you want to get out of this class?
- Tell me about your current development environment

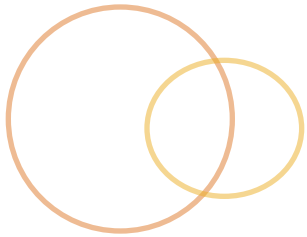
How this class works…

- Class starts whether everyone is here or not
- Break in the morning, lunch around noon (for an hour or so), break in the afternoon
- Class is done when we cover everything or our brains hurt too much to continue
- I'm flexible!
  - Ask questions if something isn't clear
  - Tell me if you already know something and we'll skip it
  - We can go hands-on at any point if something is unclear, just let me know
- There's too much to cover in too little time, but we'll try

IDEs & Editors

- Use whatever you feel most comfortable with, but I recommend:
  - WebStorm http://www.jetbrains.com/webstorm
  - SublimeText http://www.sublimetext.com
  - VIM http://www.vim.org
  - Notepad++ http://notepad-plus-plus.org

devdocs.io



- Online, searchable application for various programming language documentation
- http://devdocs.io

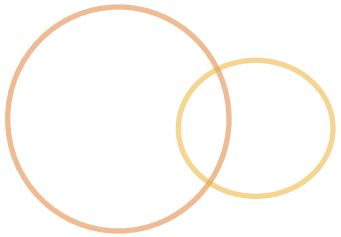
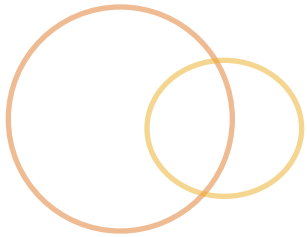
Goals for this class

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

PDF for today

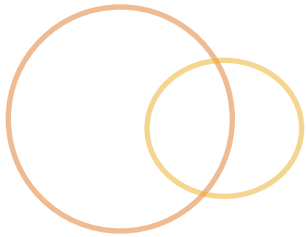
- In case you didn't quite catch something, or can't see the screen well:
  - http://bit.ly/1JOKG12

Day 1

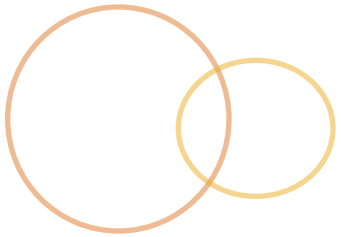

- Language recap
- Functions in JavaScript
- Object-Oriented JavaScript
- How to create modules
- Asynchronous callbacks and promises

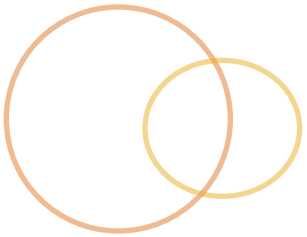
Pop Quiz!



- Let's warm up with a few exercises before we dive into the class proper

Variables



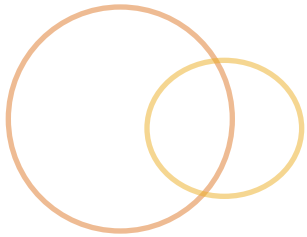


- What is logged to the console?

```
function foo() {
      console.log(b);
      var b = 1;
}

foo();
```

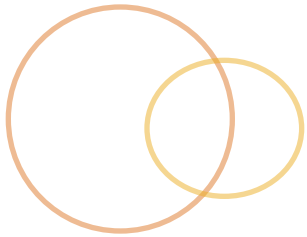
Variable scope



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

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

Callbacks



- What does this code do?

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

DOM – Refresher

- 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

DOM – Retrieval

- Using **`document`** or a previously selected element
- Returns a `NodeList`
  - **`.getElementsByTagName`**`("a");`
  - **`.getElementsByClassName`**`("fancy");`
  - **`.querySelectorAll`**`("p > span");`
- Returns a single element
  - **`.getElementById`**`("main");`
  - **`.querySelector`**`("p + p");`

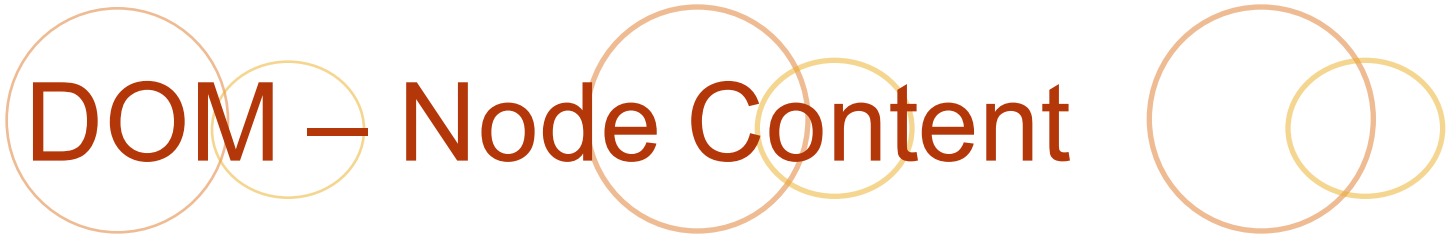
DOM – Traversal

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

DOM – 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

DOM – Node Content



- Text node content
  - `textNode.`**`nodeValue`**
- Element node content
  - `el.`**`textContent`**
  - `el.`**`innerText`**
  - `el.`**`innerHTML`**

DOM – Manipulation

- DOM manipulation
  - `.createElement("div")`
  - `.createTextNode("foo bar")`
  - `.appendChild(el)`
  - `.removeChild(el)`
  - `.insertBefore(newEl, beforeEl);`
  - `.replaceChild(newEl, oldEl)`

DOM – Element Attributes

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

DOM – Events

- 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);
```

DOM – Event Propagation

- An event triggered on an element is also triggered on all “ancestor” elements
- Two models
  - Trickling, aka Capturing (Netscape)
  - Bubbling (MS)
- W3C decided to support both
  - Starts in capturing, then bubbling
  - Defaults to bubbling
- Bubbling supports Event Delegation
  - attach an event handler to a common parent of many nodes… and parent can determine source child and dispatch as needed.

DOM – Warmup Exercise

- Fork me
  - http://jsfiddle.net/jmcneese/upn9obc7

Functions in JavaScript

- Functions are first-class objects in JavaScript, in that they are instances of the **Function** object, have state and methods, and can be passed around as variables
- Being first-class objects, they provide a number of ways that make programming JavaScript more powerful, flexible, and readable:
  - Anonymous/Lambda
  - Closures
  - IIFEs
  - Context Binding and Chaining
  - Partial Application

Functions – Special Variables

- Functions have access to two special internal pseudo-variables when invoked:
- **this** – a reference to the containing object
  - In a function statement, it refers to the global object
  - In a function expression, it refers to the most immediate containing object
  - In a constructor, it refers to the object being constructed
- **arguments** – an array-like object containing the parameters passed to the function
  - Not a real array
  - Can be converted to an array

```
var args = Array.prototype.slice(arguments);
```

Functions – Anonymous/Lambda

- Functions that are defined via an expression, or passed into another function as an argument, and not necessarily labeled
- Can be assigned to a variable
- Can be passed around
- One of the most useful and powerful features of JavaScript

```
var add = function(x, y, cb) {
      cb(x + y);
};
add(10, 20, function(sum) {
      console.log(sum); // 30
});
```

Functions – Context

- *Context* refers to the value of **this** within the function's scope
- Functions inherit the context of where they were executed, *not* where they are defined

```
var obj = {prop: "foo"};
var fn = function() {
      console.log(this.prop);
};

fn(); // undefined
obj.fn = fn;
obj.fn(); // "foo"
```

Functions – Binding

- Sometimes you want to specify the context that a function is called in

```
var obj = {prop: "foo"};
var fn = function(x, y) {
     console.log(this.prop, x, y);
};
fn.call(obj, 'bar', 'baz');
fn.apply(obj, ['bar', 'baz']);
```

Functions – Partial Application

- Sometimes you want to 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

var add10 = add.bind(null, 10);
add10(2);      // 12
```

Functions – Closures

- A **closure** is created when an inner function has access to an outer (enclosing) function's variables
- It has access three scopes:
  - Own – variables defined in its body
  - Outer – parameters and variables in the outer function
  - Global
- Gotchas
  - Closures maintain access to the outer function's variables even after the outer function returns
  - Closures store *references* to the outer function's variables, not the values of those variables
- Pragmatically, *every* function in JavaScript is a closure!

Functions – Closure Examples

- Whimsical
  - http://jsfiddle.net/jmcneese/xrv6vvkk
- Practical
  - http://jsfiddle.net/jmcneese/u54mazck

Functions – IIFEs

- **I**mmediately **I**nvoked **F**unction **E**xpression
- A function that is defined within a parenthesis, and immediately executed
- Different from closures in that variables from the outer scope are passed in, and thus the value of that variable is acted upon, not the reference
- Useful for:
  - Creating functions that need to act on the current value of a variable, rather than a reference
  - Creating and maintaining state

Functions – IIFE Anatomy

```
var x = 1;
var fn = (function(y) {
    return function(z) {
        return y+z;
    };
}(x));
x = 2;
fn(2);
```

Functions – Chaining

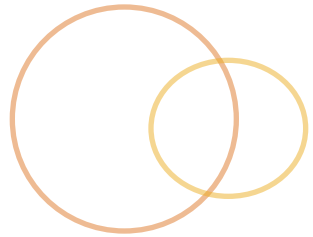
- Many functions in JavaScript return **this** when executing, enabling chained calls:

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

Functions – Chaining Example

```
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'); // Cat
```

Functions – Recap



- 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

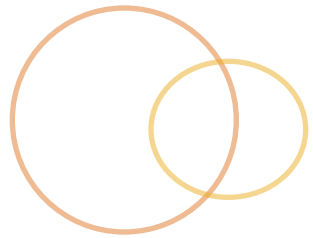
Functions – Exercise

- Fork me
  - http://jsfiddle.net/jmcneese/wguk7zdh

Object-Oriented JavaScript

- JavaScript is an inherently object-oriented language
- Prototype-based model
- Supports common OOP concepts:
  - Namespacing
  - Inheritance
  - Composition
  - Encapsulation
  - Abstraction
  - Polymorphism

OO – Namespaces



- Namespaces are easy, just create an object!
  - Object property values can be any data type, including other objects

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

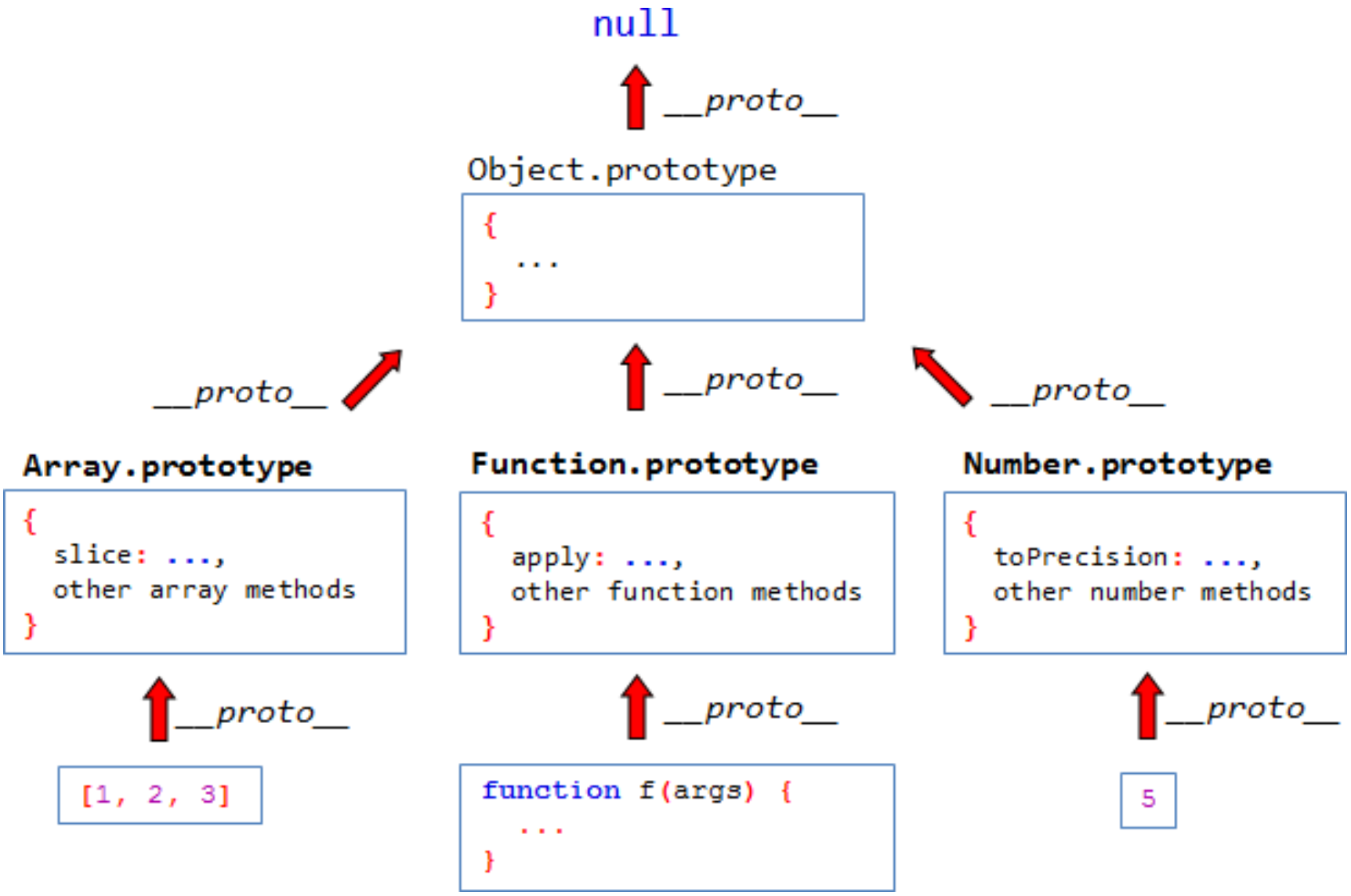
OO – Prototypes

- **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

OO – Prototype Chain

- When you request a property of an object, it checks the object, then its prototype, then the prototype's prototype, and so on…
- This is the basis of inheritance in JavaScript

OO – Prototypes Visualized



OO – prototype vs. __proto__

- **.prototype** is a property of the Function object
  - It is created when a function is defined
  - When a function is used as a constructor, it indicates the prototype of objects constructed by said function
- **.__proto__** is an instance property of an object
  - References its prototype
  - This is the *prototype chain* we referred to earlier

OO – Object Creation

- There are three ways to create an object:
  - Object literal

```
var obj = {};
```

  - Object construction

```
var obj = new Object();
```

  - Object.create

```
var obj = Object.create(Object.prototype);
```

OO – Everything is Object

- All objects inherit from other objects*

OO – Prototypal Inheritance

- In an effort to make the prototypal model more palatable, the **new** operator was added to the language in ancient times

```
var arr = new Array(1, 2, 3);
```

  - A new object is created, inheriting from **`Array.prototype`**
  - The **`Array()`** constructor function is called with the provided arguments
    - Within the constructor function's body, **this** refers to the newly created object
    - The newly created object is returned automatically

OO – Inheritance Example

- This is the recommended "native" way to implement inheritance for modern JavaScript:
  - http://jsfiddle.net/jmcneese/v4h7jyrc

OO – Inheritance Helper

- Since inheritance is programmatic in JavaScript, we can create helpers to make things easier:
  - http://jsfiddle.net/jmcneese/p2ohmuw0

OO – Why prototypes?

- Creation by copying, rather than instantiation
  - Self-describing, no need for a "blueprint"
  - Examples of objects, rather than descriptions of them
- Allows for run-time modification of a single object, or entire set of objects
- Can simulate classical inheritance as needed

OO – Encapsulation

- We've already seen that objects encapsulate data and functions that act on that data
- As of ES5, JavaScript doesn't support the concept of information hiding via visibility keywords
  - However, if we want to simulate this, we can combine object creation with an IIFE
  - http://jsfiddle.net/jmcneese/r9h79srb

OO – Immutability

- **Object.freeze**(obj)
  - Makes *obj* immutable, preventing any modification of any kind
  - http://jsfiddle.net/jmcneese/2oduvLjd
- **Object.preventExtensions**(obj)
  - Prevents new properties from being added to *obj*
  - http://jsfiddle.net/jmcneese/dL5zgzoh
- **Object.seal**(obj)
  - Prevents properties from being added to or deleted from *obj* and marks all existing properties as non-configurable, though the property values may still be changed
  - http://jsfiddle.net/jmcneese/y6hpx575
- **Object.defineProperty**(obj, propName, definition)
  - Defines (or updates) a property on *obj*
  - http://jsfiddle.net/jmcneese/zajea712

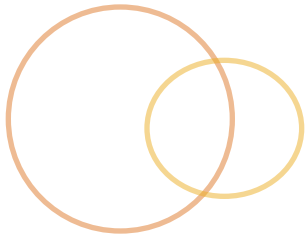
OO – Recap

- 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

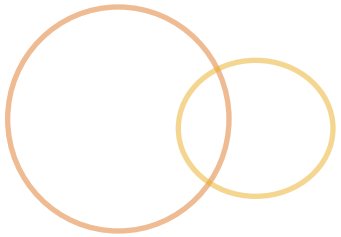
- Fork me
  - http://jsfiddle.net/jmcneese/vv55wqcu

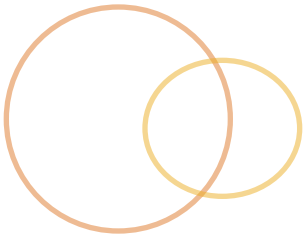
Modules



- Core OOP concept
- By definition, modules are a “technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each modules contains everything necessary to execute only one aspect of the desired functionality”
- Modules **import** functionality from other modules that **export** their functionality
- Makes life easier when dealing with large, complex systems

Modules





- As of ES5, there is no language-level construct for creating modules
- ES6 *does* include this construct, but browser support is minimal/non-existent
- Today we can use:
  - Revealing Module Pattern (**RMP**) for simple implementations
  - Asynchronous Module Definition (**AMD**) as implemented in the **require.js** library for more complex applications

Modules – RMP

- **R**evealing **M**odule **P**attern
  - We've already seen this in the form of an IIFE
- IIFE allows us to define "private" variables and use them within the scope of the closure
- Example:
  - http://jsfiddle.net/jmcneese/w83Leo50
- Gotchas
  - Fragile
  - No dependency management
  - Non-performant at scale

Modules – AMD

- **A**synchronous **M**odule **D**efinition
  - Standardized API for defining modules and their dependencies, as well as loading modules asynchronously as required
- Example using **require.js**
  - http://jsfiddle.net/jmcneese/7p1rebjp
- Pros
  - Multiple modules can be loaded in parallel
  - Perfect for web-applications
  - Dependencies can be loaded anytime, as needed
- Cons
  - Requires your application to be implemented inside require() and define() calls

Modules – ES6

- ES6 will provide native module support
- Module definitions are revealed via **export** keyword
- Modules are required via **import** keyword
- Example, if you are curious how that would look
  - http://bit.ly/1JsrcL6

Modules – Recap

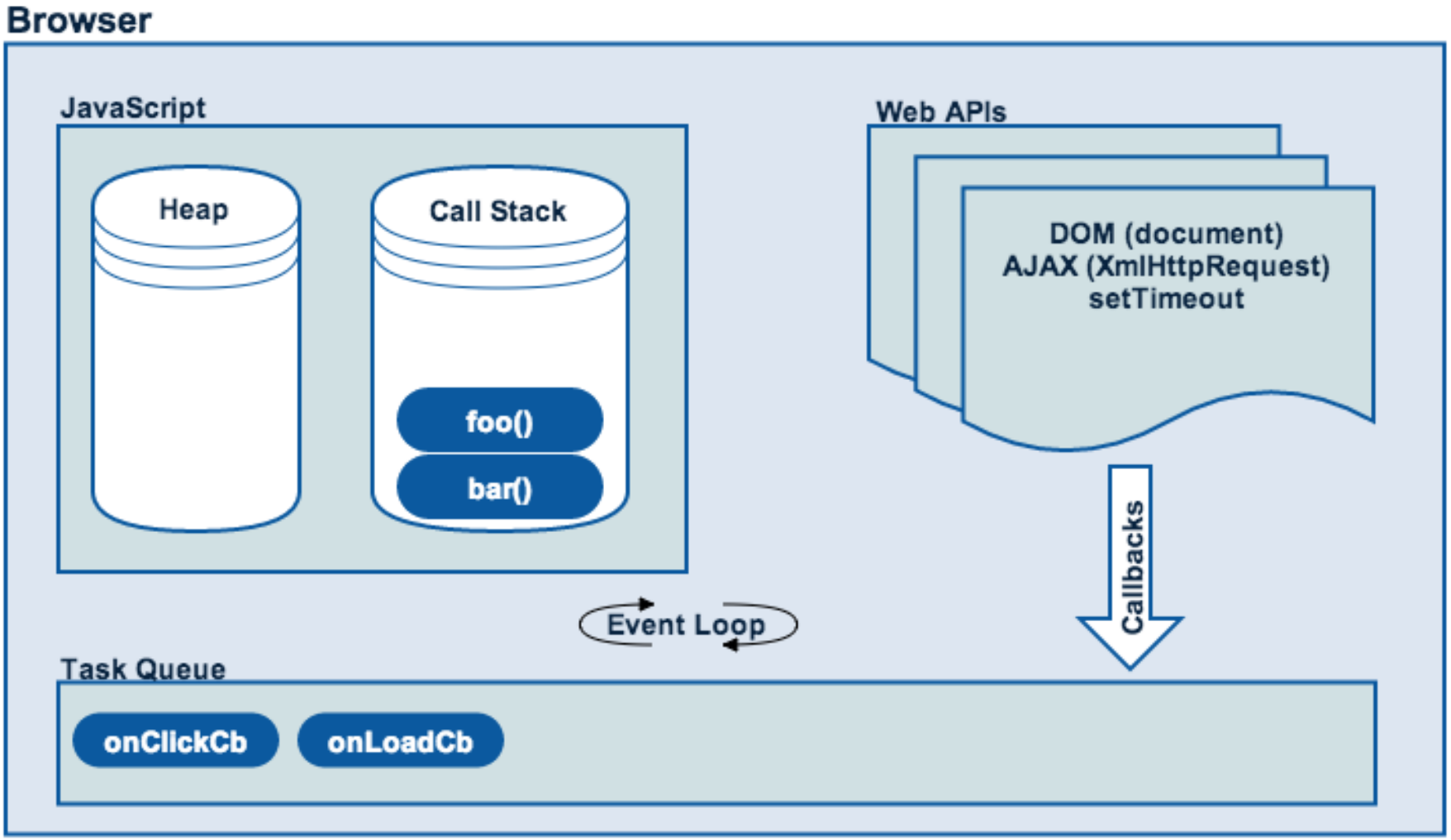
- No current language-level support for modules
- Revealing Module Pattern, which is an IIFE, can solve simple problems
- AMD, via require.js, to manage more complex or on-demand modules and their dependencies
- ES6 will make all this obsolete, whenever browser vendors decide to support it fully

Modules – Exercise

- Let's create a set of modules, using both methods
- RMP
  - http://jsfiddle.net/jmcneese/0vnv420q
- AMD
  - http://jsfiddle.net/jmcneese/o9ganz0d

Asynchronous Programming

- Does everyone know the event-loop?

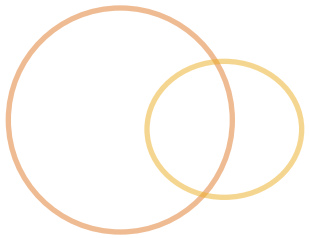


Async – Nested Callbacks

- As our applications grow over time, gaining complexity, becoming more "real-time", we one day find ourselves looking at code that looks like this:

```
async1(function(err, result1) {
        async2(function(err, result2) {
                async3(function(err, result3) {
                        async4(function(err, result4) {
                                ...
                        });
                });
        });
});
```

Async – Promises



- A **Promise** represents a proxy for a value not necessarily known when the promise is created
- Internally, it allows you to attach handlers to an asynchronous operation's success or failure
- It returns instead of the final value, the *promise* of having a value at some point in the future
- This allows code to change from continuation-passing style

```
getMyTweets(function(err, tweets) {});
```

- To one where your functions return a value which represents the eventual results of that operation

```
var promisedTweets = getMyTweets();
```

Async – 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

Async – Promise Object

- A Promise constructor takes a single argument, which is a function that has two arguments **fulfill** and **reject**
- The promise instance has a method **then**, which allows attaching handlers to the fulfill or reject events

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

Async – Pyramid of Doom

- 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) {
                });
        });
});
```

Async – 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
        });
```

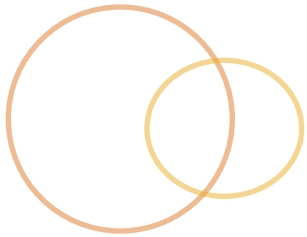
Async – Promises vs. Listeners

- A promise works a little bit like an event listener callback, except:
  - it can only succeed or fail once
  - it cannot change from succeeded to failed or vice versa
  - if a promise has resolved already, but you add another callback to it, it will call the new callback with the appropriate data
- Promises are not replacements for event listeners, unless your event only needs to fire once
  - Even then, you are likely adding unnecessary complexity to a simpler solution

Async – Promise Examples

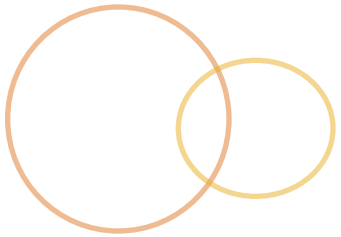
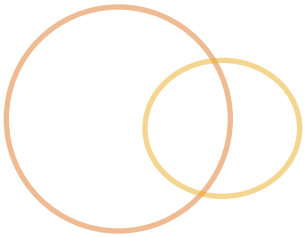
- Simple promise enabled XHR
  - http://jsfiddle.net/jmcneese/0505z09q
- Fancy parallel XHR page-building
  - http://bit.ly/1EeP0pR

Time for a pint!



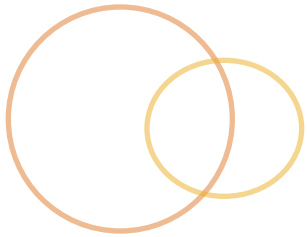
- Q&A

Day 2


- Learn how to use:
  - **bower** to manage our libraries
  - **jquery** to handle DOM and XHR logic
  - **underscore** to abstract low-level logic
  - **json-server** to create a fake REST API server
- Take a peek at a full-fledged MVC single-page application
- Create a simple single-page application using the abovementioned tools and libraries

Feedback?

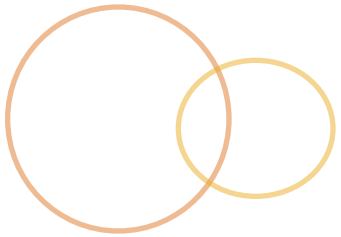
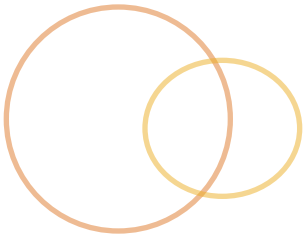


- Any unanswered questions or lingering fear/uncertainty/doubt from yesterday?

PDF for today

- In case you didn't quite catch something, or can't see the screen well:
  - http://bit.ly/1LYyBJ4

Setup


- We are going to be building things live today, so make sure you have **node/npm** installed
- Create a project directory, where we will be doing most of our work

Bower

- http://bower.io
- A package manager for the web
  - Locates, downloads and saves components you need for your application
  - Keeps track of installed packages in a manifest file
  - It's up to you to use installed packages

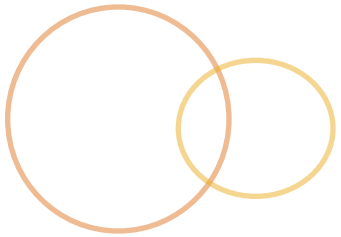
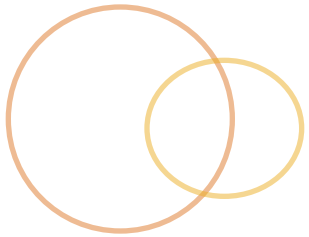
Bower – Setup

- First, we need to install it

```
npm install -g bower
```

- Add it to devdocs.io

jQuery


- http://jquery.com
- “It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.”
  - Small, fast and powerful library
  - Possibly the largest JavaScript community
  - Extensible, plugin-based
    - Beware of utterly terrible contrib code!

jQuery – Setup

- First, we need to install it

```
bower install jquery
```

- Add it to devdocs.io

jQuery – Exercise

- Let's do our very first exercise again, but this time use jQuery instead of straight DOM API
- Fork me
  - http://jsfiddle.net/jmcneese/upn9obc7
- We could easily spend several days learning about jQuery
  - Use devdocs.io
  - If you get stuck or need pointers, let me know!

Underscore

- http://underscorejs.org
- “a JavaScript library that provides a whole mess of useful functional programming helpers without extending any built-in objects”
  - Specifically designed to work alongside jQuery and Backbone (but is agnostic)
  - Provides functions to help working with collections, arrays, objects and functions, as well as various utilities

Underscore – Setup

- First, we need to install it

```
bower install underscore
```

- Add it to devdocs.io

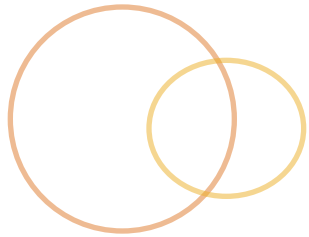
Underscore – Exercise

- Let's refactor our Collection and Item objects from yesterday to use Underscore where possible
- For those of you who didn't do Bonus B, here's a completed solution that has what you need:
  - http://jsfiddle.net/jmcneese/0Lyu0t7x
- There are hundreds of functions available, we could spend all day learning it
  - Things are named/aliased as you would expect
  - Let me know if you get stuck!
- Make sure the same tests pass for our refactored version!

Backbone

- http://backbonejs.org
- “Gives structure to web applications by providing **models** with key-value binding and custom events, **collections** with a rich API of enumerable functions, **views** with declarative event handling, and connects it all to your existing API over a RESTful JSON interface”
  - Built specifically with jQuery and Underscore in mind
  - Used by AuraJS (at least I think so!)

Backbone – Setup



- First, we need to install it

```
bower install backbone
```

- Add it to devdocs.io

Single-Page Application

- We don’t have enough time to fully or truly learn building single-page MV* applications...
- So I built you a couple to use as examples/reference for our final exercise
  - https://github.com/jmcneese/salesforce

json-server

- https://github.com/typicode/json-server
- “Full fake REST API with **zero coding**”
  - Given a json db file, json-server will automatically create routes for all the resources in the db, as well as create associations between different models

json-server – Setup

- First, we need to install it

```
npm install -g json-server
```

- Fire it up

```
json-server -w db.json
```

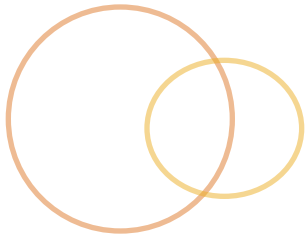
Debugging Refresher

- **Don’t use alert(), or even console.log()**
- **debugger**
  - Adding this keyword anywhere in your code will trigger a breakpoint in most browser developer tools
- Chrome Dev Tools
  - Let’s explore for a bit, if you want

Putting it all together

- Using the examples and db.json provided, and your own local **json-server**, build either:
- Gallery
  - Index: Displays clickable album links
  - View: Displays album, author and photos
  - Bonus: Be able to delete photos from albums
- Blog
  - Index: Display 10 posts
  - View: Display post, author and comments
  - Bonus: Enable pagination for index and/or comments

That's all, folks!



- Q&A