KEY CONCEPTS OF WEEK 1:
1 - Object Creation & Inheritance
2 - Closures
3 - Determining THIS
4 - The DOM

Recommended Readings:
1 - DOM Enlightenment
2 - How Browsers Work
3 - What Is The Execution Context?
4 - *Effective JavaScript* by David Herman
5 - *The Principles of Object Oriented JavaScript* by Nicholas C. Zakas
6 - *JavaScript: The Good Parts* by Douglas Crockford
7 - *JavaScript: The Definitive Guide* by David Flanagan

1. OBJECT CREATION & INHERITANCE

This week we reviewed data structures. Much of the discomfort many of you felt had little to do with your understanding of what a linked list or a queue or a hash is. In fact, most of you could articulately describe how these structures work. The data structures workshop is difficult because it forces you to leverage what we've taught you about object creation to codify your mental models. That's why the most important concept to review this weekend is object creation.

Classical Inheritance:
JavaScript Programmers often use the term 'classical inheritance' when describing a type of inheritance that emulates the class-based system of languages like Java (remember, many programming languages have the notion of 'classes' as blueprints for objects). JavaScript doesn't have classes, so the term 'Classical Inheritance' can be misleading. Objects in JavaScript are just key-value pairs, which you can create and change at any time.

However, JavaScript does have the **new** operator, which, when invoked with a function (we call functions invoked with the **new** operator **constructor functions**), achieves functionality that is very similar to the functionality produced by the class syntax of other languages. Note that the term **constructor functions** is misleading: any function can be used as a constructor function. Constructors are just functions that happen to be called with the **new** operator in front of them; they are not special types of functions; they're just regular functions that

are, in essence, simply hijacked by the use of the **new** operator in their invocation. To quote JavaScript ninja Kyle Simpson, "there's no such thing as 'constructor functions', but rather construction calls *of* functions."

So, to get back to how Java has classes and JavaScript doesn't, in Java you can do this:

```
Person adam = new Person();
```

In JavaScript, you can do this:

```
var adam = new Person();
```

This construction call of the 'Person' function makes Person seem like a class but in fact it is just a function. Keep this in mind as you study. Also remember that 'Classical Inheritance' in JavaScript refers to a host of methods by which programmers combine functions with the **new** operator in order to make object A's functionality available to object B.

The NEW Operator Does 4 Things:

1 - It creates a new object. The **new** operator essentially endows a function with the power to construct an object.

2 - It sets this new object's internal [[prototype]] property to be the constructor function's external, accessible, prototype object (every function object automatically has a prototype property that is itself an object). An object's internal [[prototype]] property creates the linkage by which method retrieval via the prototype chain occurs. There are only three ways to determine what an object's internal [[prototype]] property points to:

a. *someObject*.__proto__: this is both a setter and getter function that exposes the value of the internal [[prototype]] of an object. We can think of .__proto__ as the public link that corresponds to the private [[prototype]] link.

b. Object.getPrototypeOf(*someObject*)

c. *someObject*.constructor.prototype (if neither the constructor nor the prototype properties have been modified. This is the least reliable way to retrieve the value of the [[prototype]] property.

Check out this gist.

3 - It executes the constructor function, using the newly created object whenever **this** is mentioned (in order to pass on the **own**

properties of the constructor function to the newly created object). In other words, the newly constructed object is set as the **this** binding for that function call.
4 - It returns the newly created object, unless the constructor function returns a non-primitive value (i.e. unless it returns its own alternate object). In this case, that non-primitive value will be returned.

Every object has this internal property called *[[prototype]];* .*__proto__* is the publicly available accessor for this internal property. .__proto__ is controversially used because it is not a standard accessor. ES6 is standardizing .__proto__. The internal [[prototype]] property is set at object creation time, either with *new,* with *Object.create,* or based on the object literal ({}) which defaults to Object.

Check out the gist: [Classical Inheritance]

Prototypal Inheritance

Prototypal Inheritance does not try to emulate the class system. Objects inherit from other objects. You take an object that you'd like to reuse and then you create a second object that gets its functionality from the first one.

Most prototypal inheritance methods combine object literal notation with **Object.create()** to transfer functionality.

The Object.create() Method Does 3 Things:
1 - It creates a new object. The **Object.create() method** takes one required argument: the object that should be the prototype of the newly-created object. The second, optional argument is an object that sets properties typically fed to Object.defineProperties().
2 - It sets this new object's internal [[prototype]] property to be the constructor function's external, accessible, prototype object (every function object automatically has a prototype property that is itself an object). An object's internal [[prototype]] property creates the linkage by which method retrieval via the prototype chain occurs. There are only three ways to determine what an object's internal [[prototype]] property points to:
d. *someObject*.__proto__: this is both a setter and getter function that exposes the value of the internal [[prototype]] of an object. We

can think of .__proto__ as the public link that corresponds to the private [[prototype]] link.
e. Object.getPrototypeOf(*someObject*): this method returns the prototype (i.e. the value of the internal [[Prototype]] property) of the specified object. So *someObject* is the object whose internal prototype property is to be returned.
f. *someObject*.constructor.prototype (if neither the constructor nor the prototype properties have been modified. This is the least reliable way to retrieve the value of the [[prototype]] property.
3 - It returns the newly created object, unless the constructor function returns a non-primitive value (i.e. unless it returns its own alternate object). In this case, that non-primitive value will be returned.

If these three things look familiar it's because they are the same as 1, 2, & 4 from the list of things that the **new** operator does. Unlike the **new** operator, Object.create() does not execute a constructor function with the newly created object (thereby giving it own properties).

Check out this gist on Object.create()
Check out the gist on Prototypal Inheritance

Explaining [[Prototype]]

Perhaps the most critical operation that both **Object.create()** and the **new** operator perform is that of linking the newly created and returned object's internal [[prototype]] property to the passed in object (in the case of Object.create()) or to the constructor function's prototype property (in the case of the new operator). This link between the new object's [[prototype]] and the source object's prototype object is key. An object's internal [[prototype]] property creates the linkage by which method retrieval via the prototype chain occurs. When attempting a property access on an object that doesn't have that property, the object's internal [[prototype]] linkage defines where the [[Get]] operation will look next. This cascading linkage from object to object essentially defines a 'prototype chain' (which is an analog to a nested scope chain in many ways) of objects to traverse for property resolution. All normal objects have the built-in Object.prototype as the top of the prototype chain (like the global scope in the scope lookup), where property resolution will stop will stop if not found anywhere prior in the chain. toString(), valueOf(), and other common

utilities exist on this Object.prototype object, explaining how all objects in the language are able to access them.

If you're still uncomfortable with object inheritance after reviewing these gists, check out **ObjectPlayground.com** and explore various diagrams that they include on their page. Also check out **Principles of Object Oriented JavaScript** by Nicholas C. Zakas.

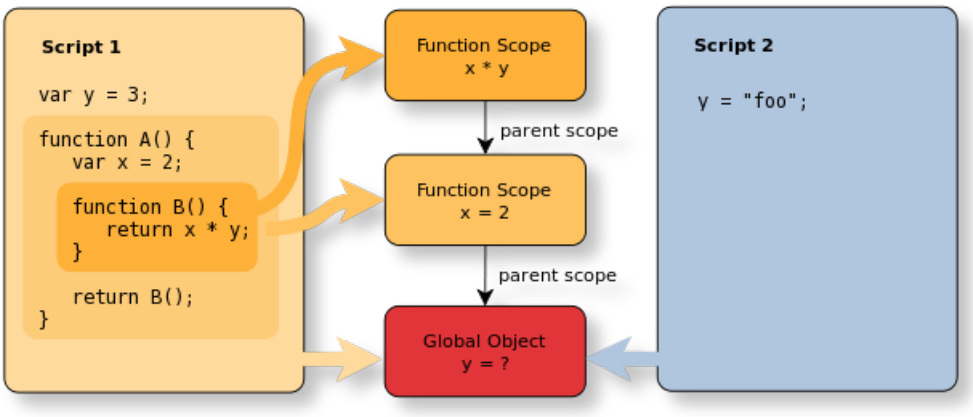
2. CLOSURES

Closures are functions that keep track of variables from their containing/enclosing scopes. In other words, closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

Three basic things to keep in mind:

1 - JavaScript allows you to refer to variables that were defined outside of the current function. This is referred to as **lexical scoping**. A function's lexical scope is determined by the scope in which it is defined. In other words, functions run in the scope in which they are defined, not the scope from which they are executed.

Check out this diagram of lexical scope:



The global object is the top level scope, which is shared by all scripts. So in this diagram, Script 2 is imported. And if Script 2 has a variable y in its global scope and Script 1 has a variable y in its global scope, whichever script is imported last sets the value of y. This is why global variables are bad: they pollute the global space and

are vulnerable to being overridden by third-party scripts you import. More on this in another concept review.

The main point here is that functions can access variables from their enclosing scopes because of the lexical scoping nature of JavaScript.

2 - Closures refer to variables defined in outer, enclosing functions ***even after*** **those outer functions have returned (returned is equivalent to 'exiting')**. Thus, closures outlive the functions that made them.

3 - Closures **store references** to variables defined within their enclosing scopes rather than copying their values. So any changes to these variables are accessible to the closure.

Another programmer way of thinking about this is that whenever a function is defined, the JavaScript engine keeps a reference to all the variables currently available in scope that the function references. Then, even if the outer function exits, those references to the variables remain and don't allow the JavaScript garbage collector to delete them. This is how JavaScript function closures keep access to the variables that are in scope when they are defined.

Check out this gist that demonstrates closure

3. DETERMINING THIS ALL THE TIME, EVERYWHERE

In order to know the value of THIS, you must inspect the invocation context, which is the location in code where a function is called (NOT where the code is declared. Recall that the lexical scope of a function is defined by where that function is declared. This is not true for THIS; THIS is controlled by the invocation context. There is absolutely NO connection between the lexical context and the invocation context which means that there is no connection between THIS and lexical scope). Only by inspecting the invocation context can we answer the question: what object is THIS pointing to?

Just to make sure we're all clear -

This is a function declaration:

```
function foo() {

      console.log('this', this)
      //this definition has nothing to do with the value of THIS
}
```

And this is how we invoke foo:

```
foo() //this invocation has EVERYTHING to do with the value of THIS
```

A function declaration is an assignment - we give **foo** a value. Simply giving a function a value doesn't invoke it! The console.log will not appear as a result of the declaration. Invoking the function triggers the console.log.

Determining the invocation context is not always as simple as locating where a function is called because certain coding patterns obfuscate the true invocation context. We can inspect the invocation context by using the call-stack (the stack of functions that have been called to get us to the current moment in execution).

Let's check out some code that demonstrates the call-stack and call-site (start by looking at the bottom where function baz is invoked!):

```
function baz() {
      //call-stack is: `baz` so, our invocation context is the global
scope

      console.og('baz');
      bar(); // the invocation context for `bar` is `baz`
}

function bar() {
      // call-stack is `baz` → `bar`

      console.log('bar');
      foo(); // the invocation context for `foo` is `bar`
}

function foo() {
```

```
      //call-stack is: `baz` →  `bar` → `foo`

      console.log('foo')
}

baz(); // the invocation context for `baz` is the window
```

By looking at the chain of function calls in order, we can visualize the call stack. The easiest ways to do this in your console are by 1) inserting the debugger and 2) setting a breakpoint in the tools. These tools will show you a list of the functions that have been called to get to that line, which is your call-stack.

Check out this gist to see an example of the **THIS** confusion before learning about the four rules for determining the value of **THIS**.

So How Does The Invocation Context Determine THIS?
Four rules govern the value that the invocation context assigns to **THIS.** We've listed these rules in order of precedence (that is, new binding overrides explicit/hard binding which overrides implicit binding which overrides default binding). Keep in mind, though, that you'll most frequently encounter 3 and 4, which are implicit and default binding, respectively. Here are 4 the rules:

1. Is the function called with **new** (a.k.a. *new binding*)? If so, **THIS** is the newly constructed object.
`var bar = new foo()`
Check out the New Binding gist

2. Is the function called with **call(), apply() or bind()** (a.k.a. *explicit/hard binding*)? If so, **THIS** is the explicitly specified object. We use explicit binding when we want to force a function call to use a particular object for the **THIS** binding without putting a property function reference on the object. **NOTE: call(), apply() and bind() are built-in JavaScript methods that explicitly set the object to which THIS points.**

`var bar = foo.call( obj2 )`

Check out the Explicit Binding & Explicit Binding w/ bind() gists and then compare Explicit Binding to New Binding and see how bind overrides call & apply

3. Is the function called with a context (a.k.a. *implicit binding*), otherwise known as **an owning or containing object**? If so, **THIS** is *that* context object. With implicit binding, most of the time, **THIS** is set by whatever object appears before the dot (obj1 in this example).

```
var bar = obj1.foo();
```

Check out the Implicit Binding gist and then see how explicit binding overrides implicit binding here.

4. Otherwise, default the **THIS** (*default binding*). Most function calls occur in the window: they're standalone function invocations. If in strict mode, the default **THIS** is set to **undefined**; if not in strict mode, the default **THIS** is set to the **global object**.

```
var bar = foo();
```

Check out the Default Binding gist

When you feel comfortable with these rules, check out some common foibles with implicit binding: Implicitly Lost Binding & Ignored This & Indirect Referencing

4. THE DOM

The DOM is the product of the browser's parsing of your HTML. It is also a tree-like object model that enables programmers to get, change, add, or delete HTML elements. When queried for nodes, the DOM returns array-like objects. Like arrays, array-like objects possess various numbered elements and a length property but they don't have any of the methods defined on the Array prototype (think of the arguments array-like object that is available to every function).

Check out this gist on converting array-like objects into arrays and read Why 'Slice' my arguments?: The Jazz Behind Array.prototype.slice() by recent Fullstack graduate Ginna Baker (**NOTE: you'll have to scroll down and click on 'older posts' to get to the blogpost**).

To understand the DOM, we have to understand the browser. You just completed selector.js which means you have a pretty solid understanding of how the DOM must be traversed recursively.

We **HIGHLY** encourage all of you to spend some time reading this article (which is far better than the embedded video):
How Browsers Work