**Javascript Refresher**

\_18.5% 7 === 4.5

\_// There are three special not-a-real-number values:

Infinity; // result of e.g. 1/0

-Infinity; // result of e.g. -1/0

NaN; // result of e.g. 0/0, stands for 'Not a Number'

\_// Type coercion is performed for comparisons with double equals...

***"5" == 5; // = true***

***null == undefined; // = true***

// ...unless you use ===

**"5" === 5; // = false**

**null === undefined; // = false**

\_charAt: to access characters in a string

\_substring to get a larger piece of a string

\_// There's also `null` and `undefined`.

null; // used to indicate a deliberate non-value

undefined; // used to indicate a value is not currently present (although

// `undefined` is actually a value itself)

// false, null, undefined, NaN, 0 and "" are falsy; everything else is truthy.

// Note that 0 is falsy and "0" is truthy, even though 0 == "0"

\_// Keys are strings, but quotes aren't required if they're a valid

// JavaScript identifier. Values can be any type.

**var myObj = {myKey: "myValue", "my other key": 4};**

\_// Object attributes can also be accessed using the subscript syntax,

myObj["my other key"]; // = 4

// ... or using the dot syntax, provided the key is a valid identifier.

myObj.myKey; // = "myValue"

\_Key of objects no need quotations

\_Can add attributes to objects by: object.newkey =”new value”

\_grade = 'B';

switch (grade) {

case 'A':

console.log("Great job");

break;

case 'B':

console.log("OK job");

break;

case 'C':

console.log("You can do better");

break;

default:

console.log("Oy vey");

break;

}

\_setTimeout(myFunction, 5000); : call after 5 seconds

\_setInterval(myFunction, 5000): called every 5 seconds

\_// When functions attached to an object are called, they can access the object

// they're attached to using the `this` keyword. The function has access to the object. “this” refers to that object

myObj = {

myString: "Hello world!",

myFunc: function(){

return this.myString;

}

};

myObj.myFunc(); // = "Hello world!

* **THIS works with how the function is called, not where it’s defined. So in order to evoke “this”, you have to call it with a function**

**READ THIS CAREFULLY AFTER WAKING UP:**

// What this is set to has to do with how the function is called, not where

// it's defined. So, our function doesn't work if it isn't called in the

// context of the object.

var myFunc = myObj.myFunc;

myFunc(); // = undefined

// Inversely, a function can be assigned to the object and gain access to it

// through `this`, even if it wasn't attached when it was defined.

var myOtherFunc = function(){

return this.myString.toUpperCase();

}

myObj.myOtherFunc = myOtherFunc;

myObj.myOtherFunc(); // = "HELLO WORLD!"

// We can also specify a context for a function to execute in when we invoke it

// using `call` or `apply`.

var anotherFunc = function(s){

return this.myString + s;

}

anotherFunc.call(myObj, " And Hello Moon!"); // = "Hello World! And Hello Moon!"

// The `apply` function is nearly identical, but takes an array for an argument

// list.

anotherFunc.apply(myObj, [" And Hello Sun!"]); // = "Hello World! And Hello Sun!"

// But, `call` and `apply` are only temporary. When we want it to stick, we can

// use `bind`.

var boundFunc = anotherFunc.bind(myObj);

boundFunc(" And Hello Saturn!"); // = "Hello World! And Hello Saturn!"

// `bind` can also be used to partially apply (curry) a function.

var product = function(a, b){ return a \* b; }

var doubler = product.bind(this, 2);

doubler(8); // = 16

// When you call a function with the `new` keyword, a new object is created, and

// made available to the function via the `this` keyword. Functions designed to be

// called like that are called constructors.

var MyConstructor = function(){

this.myNumber = 5;

}

myNewObj = new MyConstructor(); // = {myNumber: 5}

myNewObj.myNumber; // = 5

\_Javscript: Object-oriented language

Q: How to use “.apply?”

A: “apply” takes an array for argument

Math.min(42,6,27) ===6

BUT var a = [42,6,27]

Math.min(a) doesn’t work

BUT Math.min.apply(Math, [42, 6, 27]); (or ‘self’) does

**\_”New” and “This” go together if I want to make available to the function via the “this” keyword**

Ex:

var MyConstructor = function(){

this.myNumber = 5;

}

myNewObj = new MyConstructor(); // = {myNumber: 5}

myNewObj.myNumber; // = 5

**PROTOTYPE:**

// Every JavaScript object has a 'prototype'. When you go to access a property

// on an object that doesn't exist on the actual object, the interpreter will

// look at its prototype.

// Some JS implementations let you access an object's prototype on the magic

// property `\_\_proto\_\_`. While this is useful for explaining prototypes it's not

// part of the standard; we'll get to standard ways of using prototypes later.

var myObj = {

myString: "Hello world!"

};

var myPrototype = {

meaningOfLife: 42,

myFunc: function(){

return this.myString.toLowerCase()

}

};

myObj.\_\_proto\_\_ = myPrototype;

myObj.meaningOfLife; // = 42

// This works for functions, too.

myObj.myFunc(); // = "hello world!"

// Of course, if your property isn't on your prototype, the prototype's

// prototype is searched, and so on.

myPrototype.\_\_proto\_\_ = {

myBoolean: true

};

myObj.myBoolean; // = true

// There's no copying involved here; each object stores a reference to its

// prototype. This means we can alter the prototype and our changes will be

// reflected everywhere.

myPrototype.meaningOfLife = 43;

myObj.meaningOfLife; // = 43

\_For In is to itierate over properties of an object, can traverse up the prototype chain until it sees a null prototype

\_But if I don’t wanna traverse up, then use ‘hasOwnProperty’

ex:

for (var x in myObj){

console.log(myObj[x]);

}

///prints:

// Hello world!

// 42

// [Function: myFunc]

// To only consider properties attached to the object itself

// and not its prototypes, use the `hasOwnProperty()` check.

for (var x in myObj){

if (myObj.hasOwnProperty(x)){

console.log(myObj[x]);

}

}

To create a prototype, use a “.prototype” on the Constructor

Ex:

MyConstructor.prototype = {

myNumber: 5,

getMyNumber: function(){

return this.myNumber;

}

};

var myNewObj2 = new MyConstructor();

myNewObj2.getMyNumber(); // = 5

myNewObj2.myNumber = 6

myNewObj2.getMyNumber(); // = 6

\_You can’t modify strings but you can add to a string

Like, if you wanna get the first character of a string:

String.prototype.firstCharacter = function(){

return this.charAt(0);

}

"abc".firstCharacter();

\_