**Functions**

* **standard function definition**           function myFunct(*parameters)*{ ....*code*....  }
* **anonymous functions** - e.g. passed as a parameter  
            myObject.flip = function( *parameters* )  { ....*code*....  }        
   //also called a **function expresssion/literal**You can find more examples at <http://helephant.com/2008/08/23/javascript-anonymous-functions/>
* **assigning a function to a variable** **(or attribute)**           var hooHah =  function( *parameters* )  { ....*code*....  }    or  
             var hooHah =  function foo( *parameters* )   { ....*code*....  }
* **IEFE or Immediately Executable Function Expressions**.  
             When you wrap a function expression in ( ) and put a () at the end it is immediately executable  
    
               (function( *parameters* )  { ....*code*....  }  ) ()  
    
               The ending () may be either inside or outside of the grand ( )

(function () {

})();

         Of course, if you want to capture a returned value you need to assign the result of the IEFE to a variable.

Anonymous functions take getting used to but they are quite common - either as a method in an object or as a callback function.

Note: When we define a method as part of an object (either with a function expression or by using a previously defined function) the name/key for the method has no (), but when we call it we use parens.  
  
 function Point(x\_value, y\_value) {  
 this.x = x\_value;  
 this.y=y\_value;  
 this. reflectAboutXAxis = function(){ this.y = -this.y;}  
 }

We may now create new Points:  
  
 origin = new Point(0,0);  
 upperRight = new Point(10, 15);

upperRight.reflectAboutXAxis()

Try it yourself: In the console of Chrome or Firefox type:

function Point(x\_value, y\_value) {  
 this.x = x\_value;  
 this.y=y\_value;  
 this. reflectAboutXAxis = function(){ this.y = -this.y;}  
 }  
  
 origin = new Point(0,0);  
 upperRight = new Point(10, 15);

console.log(upperRight.y)

upperRight.reflectAboutXAxis()

console.log(upperRight.y)

Now add a new method to upperRight. The method should be named double and it should multiply both the x and y coordinates by 2. Use an anonymous function.   
Then call it and see what the values are for upperRight.x and upperRight.y

Recall fro Part 1 of these notes that you know that every object has a prototype. So, you also know how to add a method to all Points by using Point.prototype.  
Add a new method expand to all Point. The method should take one parameter m and multiply both the x and y coordinates by m.

**NOTE: We added the double method to only upperRight, but, via the prototype, we added the multiply method to every Point.**

An aside. If I want to generate a subclass of a class I can use object.create  
Example: Point3D = object.create(Point);  
 Now we can add a z-coordinate or methods. This is very advanced, but if you wish to learn more then <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create> will show you how to do this, using the **call** or **bind** method**.**

**variables**

In JavaScript (prior to ES6) there are only global variables and variables with function-scope.  
  
Variables with function scope are declared inside a function with the reserved word **var.** Thus in  
 function myFunc() {  
 var x =7; //x is local to myFunc  
 y=8; //y is global  
 :  
 }  
Variable definition are **hoisted** – that is the definitions are moved to the top of the function code, but their value is **undefined** until the code assigns a value to the variable.

An aside: In ES6, **let** and **const** give you variables and constants with block scope and they are *not* hoisted.

**arguments**

Every function comes with a property arguments, which is an array-like object.  
(You can index through it, but it doesn’t have sort() etc. methods).   
Here is an example from <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments>

function func1(a, b, c) {

console.log(arguments[0]);

// expected output: 1

console.log(arguments[1]);

// expected output: 2

console.log(arguments[2]);

// expected output: 3

}

func1(1, 2, 3);

In other words, inside any function there is a property (acts like a variable; you don’t need to use *this*) arguments[] which you may use to get a hold of any and all the arguments which were passed into the function.

The arguments property would be useful for functions with a variable number of arguments (e.g. add this list of numbers) and in some other abstruse contexts. You are unlikely to need to write code with arguments, but you may need to read such code.

**Calling a function:**

* **standard function definition**           function myFunct(*parameters)*{ ....*code*....  }    
   Call with y = myFunc(*any* *needed arguments)*  
   Examples   
   function gcd(a, b) {…//has a return value….};  
   y = gcd(70, 32);  
     
   function showTemp() { //no return value…..};  
   showTemp();
* **anonymous functions/function expression** - e.g. passed as a parameter or assigned to an object as a method:  
            myObject.flip = function( *parameters* )  { ....*code*....  }      
   Call with  myObject.flip(*any* *needed arguments)*
* **assigning a function to a variable** **(or attribute)**           var hooHah =  function( *parameters* )  { ....*code*....  }    or  
             var hooHah =  function foo( *parameters* )   { ....*code*....  }     
   Call with hooHah(*any* *needed arguments)*  
   Examples   
   var hooHah = function(a, b){…//has a return value….};  
   y = hooHah(70, 32);  
     
   var hooHah = function showTemp() { //no return value…..};  
   hooHah();
* **IEFE or Immediately Executable Function Expressions**.  
             When you wrap a function expression in ( ) and put a () at the end it is immediately executable  
    
                      function( *parameters* )  { ....*code*....  }    
    
               The ending () may be either inside or outside of the grand ( )

(function () {

})();  
Example: Suppose that x and y are global variables

z = (function(){return 2\*x – y)})() //z gets the value of 2x-y

**Of this and that:**

* **Remember that every object (including every function) has atwo important properties:  
   prototype – this points to the prototype of the object  
   this points to the object itself.**
* **Any constructor function (either defining properties or methods in it – i.e. in the object it constructs) makes use of this:**
* Example: function Point(xx, yy) {  
   this.x = xx;  
   this.y = yy;  
   this.reflectAboutXAxis = function(){this.y \*= -1;}
* **A constructor function returns this – i.e. a pointer to the object it has just created, which, of course, you will want to assign to a variable.**Example: origin = Point(0, 0);
* **Whenever you have an object and (inside that object) you call a method of that object, the value of this is the object.**For example, in the object upperRight, there is a method reflectAboutXAxis. In the method, the value of this is upperRight.

**Nested functions and other intricasies in using this**

* **A function may create and return another function.** We’ll see this in a minute.
* Sometimes **this** can get us into trouble.  
  The issue is that the value of **this** depends on how/where things are called from.
* [**http://helephant.com/2009/11/29/javascript-method-context/**](http://helephant.com/2009/11/29/javascript-method-context/)Here is a clear simple example to work through. (There are more pages about functions at <http://helephant.com/2008/08/17/how-javascript-objects-work/> )  
    
  The bottom line:  
    
  **When a function is a method (i.e. belongs to an object) and that method is called from that object, then *this* refers to that object.  
    
  When a function is not called from an object the *this* refers to the Global Object.**
* One of the best explanations I know is at <https://www.youtube.com/watch?v=JduQUNn7L4w> and you should watch it (several times) on your own.  
  At minute 7 there is the following code:

window.onload = function() {  
 var button = getElementById(“button”);  
 button.onclick = addBanana;  
  
function addBanana(){  
 numBananas++;  
 this.innerHTML = “Add a banana (“ + numBananans + “)”;

In the first three lines we have asked that a function be defined after the page has finished loading. (When we get to jQUery we will have a better way to do that.)

What does that function do? It add an event handler, for the click event, to button.

Please notice that we are assigning to the onclick event handler a function, addBanana, which is defined elsewhere. (In JS functions are hoisted, so it doesn’t matter that the code for addBanana is later on the page the the code for the onload event handler.)

In the next three lines, innerHTML is used to change the the text inside whatever **this**  referes to. We will watch minutes 7-11 together, but it is worth watching again (& again) on your own.

* Another clear description of lexical scoping – which is what is at issue here – is at <https://alistapart.com/article/getoutbindingsituations> and we’ll work through that (but not the part about frameworks)  
    
  Just above “Look, Ma…” the article introduces the useful trick of  
   var that = this;  
  which allows you to pass the value of this to another function.
* <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures> has another clear explanation (ignore the ES6 material) and some nice code to play with in JSFiddle.  
    
  In this set of examples the makeAddr function **returns another function.  
    
  Please notice that even though the outer (makeAddr) function finishes its execution, when that function is returned, it has access to all the variables in the environment (context) in which it was created – including any variables local to makeAddr.  
    
  This is called a closure.**Try the following code:  
   function outer() {  
   var outerVar = 10;  
   function inner(x) {  
   return x\*outerVar;  
   **}**

return inner;  
 }  
 magic = outer();  
 console.log(magic(5));

In this example, outer creates a function (inner) which it returns.  
We then call outer() and assign what it returns (namely the function inner) to magic. Finally we call magic (which is inner() ) on the value 5.

* This can get hairy – see <https://stackoverflow.com/questions/7629891/functions-that-return-a-function> and its important message that:  
   **“Assigning a variable to a function (without the parenthesis) copies the reference to the function. Putting the parenthesis at the end of a function name, calls the function, returning the functions return value.”**To test this, go to the console and type  
   newMagic = outer;  
  You will see that the console prints out the code for outer – i.e. newMagic is just a reference to (another name for) outer.   
    
  Now type (in the console)  
   oldMagic = newMagic()   
   The console responds by showing the code for inner:

*ƒ inner(x) {*

*return x\*outerVar;*

*}*

That is, just as when we made magic be what outer() returned, now oldMagic   
 is what newMagic (aka outer) returns.

Finally ask for:  
 console.log(oldMagic(7))

And you will get the expected value of 70.

* **To summarize (yet again) here is how Spencer Tipping phrases it in “JavaScript in Ten** **Minutes”):**

**3.3 The meaning of** this **(the egregious disaster)**

One would think it is a simple matter to figure out what this is, but it’s

apparently quite challenging, and Javascript makes it look nearly impossible.

Outside of functions (in the global scope, that is), the word this refers to the

global object, which is window in a browser. The real question is how it behaves

inside a function, and that is determined entirely by how the function is called.

Here’s how that works:

**1.** **If the function is called alone**, e.g. foo(5), then inside that function’s body

the word this will be equivalent to the global object.

**2. If the function is called as a method**, e.g. x.foo(5), then inside that

function’s body the word this refers to the object, in this case x.

**3. If the function starts out as a method and then is called alone:**

var f = x.foo;

f (5);

then this will be the global object again. Nothing is remembered about

where f came from; it is all determined right at the invocation site.

**4. If the function is invoked using apply or call**, then this points to whatever

you set it to (unless you try to set it to null or undefined, in which

case it will be the global object again):

var f = function () {return this};

f.call (4) // => 4

f.call (0) // => 0

f.call (false) // => false

f.call (null) // => [object global]

* **Bottom line:**
  + **We can write a function outer() which returns a function inner that it creates .**
  + **We can assign that function which is being returned to a variable  
    magic by coding  
     var magic = outer();**
  + **We can then call magic like any other function – but beware of what happens to this (see below).**
  + **magic has access to all of outer’s local variables (& any other variables that outer had access to) even though outer is done executing.**
  + **If you want to use the this from outer you should introduce, in outer, a local   
     var that = this  
    or, better yet, by using call, apply, or bind.  
      
    This issue can also arise when you call a function (such as magic) from a global context.**
  + **There is also a well know problem using this and loops –** see <https://j11y.io/javascript/closures-in-javascript/> or Kyle Simpson books – **and with losing the context when you use timeouts or callbacks.**
* **Interesting subtleties:**Per  <https://stackoverflow.com/questions/4270388/what-is-the-difference-between-assigning-a-function-via-this-vs-prototype>

Functions assigned to the prototype will be shared by all instances; functions assigned in the constructor will have a separate function object per instance.

Also, functions assign in the constructor can use the constructor's variables and parameters.

More combinations and permutations on these ideas is at <https://stackoverflow.com/questions/310870/use-of-prototype-vs-this-in-javascript>

* **Examples of how you can find this with a global reference and solutions.**
  + Remember our example:  
    function Point(x\_value, y\_value) {

                            this.x = x\_value;

                            this.y=y\_value;

                            this. reflectAboutXAxis = function(){ this.y = -this.y;}

                           }

origin = new Point(0,0);

upperRight = new Point(10, 15);  
  
 If we try to say   
 function dist(u, v) {return Math.sqrt(u\*u + v\*v)} //this is fine  
 upperRight.lngth = dist(this.x, this.y)  // I get NaN   
  
 In the last line we have the global this!  
  
 ***Solution #1 - use the prototype for Point***  
  
 Point.prototype.lgth = dist(this.x, this.y)  
  
 ***Solution #2 – use call*** – please notice that the arguments for call are first the name of the object we are binding to, and then the arguments for the function we are calling on.  
  
 upperRight.d = dist.call(upperRight, this.x, this.y)

* Another example using call:  
  This example is from Chapter 2 of <https://github.com/getify/Functional-Light-JS/tree/master/manuscript>

function sum() {

    return this.x + this.y;

}

var context = {

    x: 1,

    y: 2

};

sum.call(context);    //3

* Should you want to read further:  
  + The most meticulous discussion of these issues (but it’s dense b/c it’s intended for professionals) is in Kyle Simpson’s books. They are available free – both on-line and (of course) via the ACM Safari site.  
      
    They are my go-to resource for subtleties.  
      
    The series is called “You Don’t Know JS” and they are all available at <https://github.com/getify/You-Dont-Know-JS> and in order they are:
    - Up and Going
    - Scope & Closures
    - this & Object Prototypes
    - Types &Grammer
    - Async & Performance
    - ES6 & Beyond

You are most interested in the second and third (and maybe the last).  
Warning: These are complex books – if you took the Computer Languages course, that will help.  
  
In the Scope & Closures book <https://github.com/getify/You-Dont-Know-JS/tree/master/scope%20%26%20closures> you are most interested in Chapter 2 (up to but not including the part on Cheating Lexical Scope) and Chapters 4 (on hoisting) and 5 (on closures.) If you wish we can plow thru this together.  
  
In the this & Object Prototypes book <https://github.com/getify/You-Dont-Know-JS/tree/master/this%20%26%20object%20prototypes> you are most interested in Chapters 1 and 2.

* + As usual, a very basic introduction is at <https://www.w3schools.com/js/js_function_invocation.asp>
  + <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call> describes **call()**  and apply().  
    These two functions are the same except that apply gets its arguments in the form of an array, and call as a parameter list.)
  + <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind> describes **bind**().
  + <https://www.youtube.com/watch?v=c0mLRpw-9rI> puts these together in examples of increasing compleity, but with clear explanations (especially for call and apply.)  
    .
  + <http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html> has a fairly straightforward description of hoisting.
  + <https://scotch.io/@alZami/understanding-this-in-javascript> has a good explanation of *this*.
  + More on prototypes at <http://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/> There is some use of ES6 (the appearance of **let** is the giveaway), but it is still very useful.
  + Also <https://blog.bitsrc.io/what-is-this-in-javascript-3b03480514a7> on this and call
  + <https://medium.com/@tonero91/javascript-this-keyword-a-beginner-s-approach-5c7d6a428743> explains that with timeouts & callbacks you can lose the this context.
  + <https://medium.com/@ajmeyghani/javascript-functions-a-pocket-reference-d42597ceb496> and
  + [https://medium.com/@ajmeyghani/javascript-prototypes-a-pocket-reference-d88f550ffce3](https://medium.com/@ajmeyghani/javascript-prototypes-a-pocket-reference-d88f550ffce3  %20) although both have some use of ES6.