**1. On the DOM**

     The arrays (notice the plural names)

document.images[ ] keeps track of all your images,
document.forms[ ] keeps track of all your forms,
document.links[ ] keeps track of all your links, and
document.anchors[ ] keeps track of all your anchors.

document.forms[i].elements[] keeps track of the elements in the
    i-th form.

**2. How to reference the elements**

* Using a numerical index  document.images[0]
* Using the id of the image  document.images[“myLogo”]

**3. Useful methods:**

The document object comes with three very useful methods:

·        **getElementById( )**returns the element with the id named in quotes

·        **getElementByName ( )**returns the element with the name given in qutoes

·        **getElementsByTagName( )**returns an array of all elements with the specified tag
             Notice that Element**s**is plural in the name for this method – because it
             returns a whole array of elements.  For example,
             getElementsByTagName(‘h1’) returns an array of all your h1 headings.

**4. Review "The DOM in summary"**

**5. Objects and Prototypes**

    **Array** is a built in object.  You may create new ones in several ways:

              primesTo20 =[2, 3, 5, 7, 11, 13, 17, 19];   //ARRAY notation;

 // note the SQUARE brackets.

              primesTo20 = new Array(2, 3, 5, 7, 11, 13, 17, 19);

 //Passing PARAMETERS to the Array
 //constructor function

              primesTo10 = new Array();   //defining an array and then filling it

              primesTo10[3] = 7; etc.

Arrays are indexed from 0 to whatever and have a .length property.
You can index thru an array.

An array's elements may be of different types - this is very special - and may even be functions.

An array is passed by reference, so if you want to make a copy of an array you must make a deep copy.

**Associative arrays may be indexed by strings (including integers).**
We refer to these as key-value pairs.

You may create an associative array in several ways:

myReferences = {advisor: “Menzin”, helpdesk: “Borque”, summer2011: “Bill Gates”};

This is often written as
 myReferences = {
                advisor: “Menzin”,
                helpdesk: “Borque”,
                summer2011: “Bill Gates”
                 };

This should look familiar to those of you who saw JSON (JavaScript Object Notation) in Database.
It is also the same idea as a dictionary in python (or hash in php)

You may also define an associative array with:

        myReferences = new Object();
       myReferences.advisor = “Menzin”;   etc.

Either way you may get to the values for a particular key by using either dot or bracket notation:

       advisorName = myReferences.advisor;

        advisorName = myReferences[‘advisor’];     //note use of SQUARE
 //brackets.

**As with any object in JS:**

* the value associated with some key may be changed after the object is created (i.e. assign a new value)
* more values may be added to the object after the object is created- e.g.
   myReferences.thesisAdvisor ="Grace Murray Hopper";
* similarly, a function may be assigned to a key  (see more below about functions)
* a new variety of objects may be created with a function:

function Point(xx, yy) {
   this.x = xx;
   this.y = yy'
}

It is a common but deplorable practice to use the same name for the parameters and the keys/property names- e.g. e.g.
function Point(x, y) {
   this.x = xx;
   this.y = yy'
}
* You may then create lots of new Points with the constructor function:
origin = new Point(0,0);
upperRight = new Point(10, 15);
* Every object has a **prototype.**
* **You can add to the prototype of ALL Points, including those already created with**Point.prototype.color = "";
This adds the empty string as the value of the new attribute color.   You can then re-set it in the usual way.
   origin.color ="red"
   upperRight.color = "blue"
* If I use Point to create a new object Point3D, which makes use of Point, then when looking for an attribute, JS will go up the prototype hierarchy.  (See Crockford for how to do this.) This is referred to as protypical inheritance.
* **Comment on syntax:**
When you define an object you use
   key **:**value

 whether the value is a property or a method.

When you re-assign a value you use =

   myObject.key = value; or myObject[‘key’] = value

For example:

 myAssocArray={'key1':'value1',
               'key2':function(){alert("HelloMethod");}

              };

myAssocArray['key1'] = 'newValue';

* How to iterate thru an associative array:
See Object Problems in <https://www.i-programmer.info/programming/javascript/1441-javascript-data-structures-the-associative-array.html?start=1> and <https://www.quirksmode.org/js/associative.html> for careful
* You can iterate thru an associative array as follows:
 for (var key in myAssocArray){
 var value = myAssocArray.key
 //do stuff
 };
* The problem

**Object Problems**

There is a problem with the fact that the associative array is used as JavaScript's object construct. When you create an associative array you are creating the simplest JavaScript object but this comes with some standard properties. This is mostly fine is you want to use the associative array as an object but not so good if you want to use it as an associative array. What happens is that you get an associative array that already has some key value pairs stored in it.  So for example if you are trying to discover if an associative array is empty you will find that even when you think it should be it isn't.

This is a particular problem when you want to iterate through the array using a for loop:

for(var key in array) {
 do something with key and array[key]
};

this loops through all of the properties of the object including those that were derived from Object. The usual solution is to use the hasOwnProperty method which returns true if the key is one you actually defined.

for(var key in array) {
 if(array.hasOwnProperty(key)
    do something with key and array[key]
};

NOTE: When you iterate through an associative array the order is not guaranteed.

**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**
* **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)

Coming next: this, the execution context, and closures