**Learning JavaScript (JS), By: Gabriel Moreira**

More info at: <http://www.codecademy.com/glossary/javascript>

Interactive exercises at: <http://www.codecademy.com/tracks/javascript>

**Commands:**

* **Confirm(‘your text here’);**
  + Returns boolean
  + This creates a pop up box your screen with the selected message.
    - User presses “OK” = true
    - User presses “Cancel” = false
* **Prompt(“you text here”);**
  + Returns a String
  + This creates a pop-up box that lets the user input information
    - User writes something and presses ok = returns the string of what they wrote
    - User doesn’t write anything and presses ok = returns an empty string (“”)
    - User presses cancel = returns null
  + You can convert the string passed in. e.g.
    - var answer = prompt("Question to the user").toUpperCase();
* **Console.log()**
  + Prints a string to console
  + This is JS System.out.println() Java equivalent
    - E.g console.log(2\*5) -🡪 prints 10 on the console
  + You can add convert string elements to your log as well
    - E.g. var condition = false;
    - console.log("Anything I wanna say here " + String(condition) + "!");
      * This prints this message
        + Anything I wanna say here false!
* **isNan()**
  + If you call isNaN on a data type, it checks to see if that data type is not a number.
    - isNaN('berry'); // => true
    - isNaN(NaN); // => true
    - isNaN(undefined); // => true
    - isNaN(42); // => false
    - if you call isNaN on a string that looks like a number, like '42', JavaScript will try to help by automatically converting the string '42' to the number 42 and return false (since 42 is a number).
* **Switch & case**
  + Avoids abundance of if/else statements
  + switch allows you to preset a number of options (called case), then check an expression to see if it matches any of them. If there's a match, the program will perform the action for the matching case; if there's no match, it can execute a default option.
  + Syntax:
    - switch (*/\*Some expression\*/*) {
    - case 'option1':
    - *// Do something*
    - break;
    - case 'option2':
    - *// Do something else*
    - break;
    - case 'option3':
    - *// Do a third thing*
    - break;
    - default:
    - *// Do yet another thing*
    - }

e.g.

switch(movie) {

case "Toy Story 2”:

// return review here

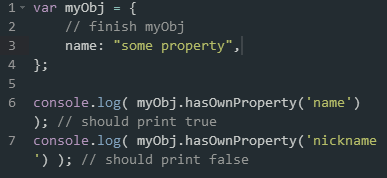
case "Finding Nemo":

// return review here

default:

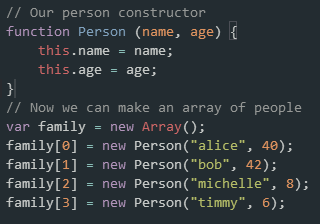
// code to be executed if a movie name is not found

}

* **Typeof**
  + Tells us the data type of a variable
  + E.g
    - var someObject = {someProperty: someValue};
    - console.log( typeof someObject );
    - this would print out “object”;
* **hasOwnProperty**
  + This lets us know if an object has a particular property.
    - It returns true or false, based on whether an object has a certain property (aka local variable or method for this object)
  + E.g
    - * 

**How to initialize**

* **To initialize an number** (aka int/double/long) just write a number with no quotations around
* **To initialize a string**, write the string using double quotes around it (“”)
* Any variable is declared as **“var”**
  + var n = 30
  + var a = “Hello”
    - “hello”.length = 4. Note that you don’t have () after length for strings in JS
  + var b = false;
* On for loops you don’t need to initialize the variable type UNLIKE JAVA
  + E.g for(I = 0; I < 4; i++){--- your code here--------}
* **To initialize an array**: var arrayName = [data, data, data,…., data];
  + You can put multiple different data types in one array. Like a Array[Object] or Array[E]
    - Eg arrayName = [String, number, number, number, String];
      * Console.log(arrayName) is now going to print this
        + [‘String’, number, number, number, ‘String’]
      * arrayName[0] = String, arrayName[2] = number
  + to add elements to an array do arrayName.push(data To Add);
    - this makes it like and arrayList of Java
    - you can initialize and array as var arrayName = [];
      * Note that I never gave a size to arrayName. Once I push things to arrayName, its size increases, just like and ArrayList
        + newArray = [];
        + newArray.push('hello');
        + newArray[0]; // equals 'hello'
  + You can also make more complex arrays of objects and use them like the below example
    - Note that the image below also shows how to add elements to an array without using push

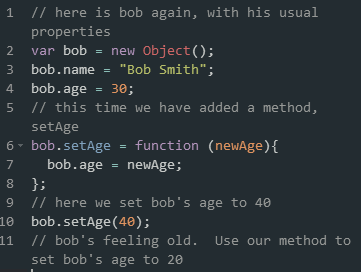
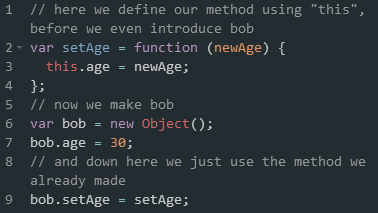
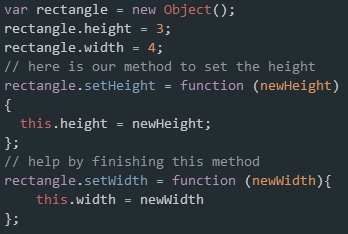
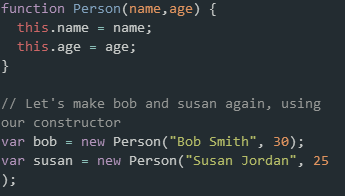
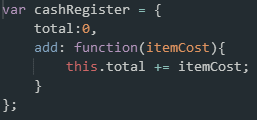
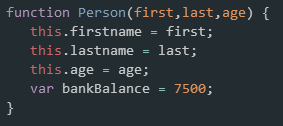
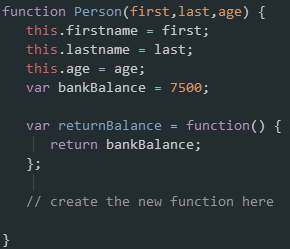


* A **for each loop** in JS is initialized as his KNOW AS A **For In loop** in JS)
  + For(var varName in iterableObject){--------------code here--------------}
    - for (var key in object) {
    - *// Access that key's value*
    - *// with object[key]*
    - }

**Object Oriented Design/Programming**

* **Function** in JS is something very similar to a method in JS, but not quite!
  + TL;DR: A method is on an object & a function is independent of an object.
    - function is a piece of code that is called by name. It can be passed data to operate on (ie. the parameters) and can optionally return data (the return value).
    - All data that is passed to a function is explicitly passed.
    - A method is a piece of code that is called by name that is associated with an object. In most respects it is identical to a function except for two key differences.
      * It is implicitly passed the object for which it was called
      * It is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data)
  + Initialized as e.g
    - Var nameOfmethod = function(Parameters e.g: input1, input 2,…., input n){ --------------------------------------- CODE FOR THE METHOD GOES HERE----------------------};
      * If the function is void, you can exclude the var e.g. nameOfmethod = function(Parameters ){);
    - Function nameOfMethod(Parameters ){)
      * This is also the same syntax of when you return a number
      * Much more similar to java syntax
  + A function with a return value can return any type of primitive data.
    - Unlike java where public String method() will always returns a string or null, in JS function functionName() can return both a string or Boolean in the SAME function!!
* **To initialize objects** you use var objectName = { ---------code here--------------};
  + THIS IS **OBJECT LITERAL NOTATION**
    - * var myObject = {
        + key: value,
        + key: value,
        + key: value
      * };
  + E.g
    - * var myObj = {
      * type: 'fancy',
      * disposition: 'sunny'
      * };
      * var emptyObj = {};
  + Note that the key could also me a method
    - methodName: function(parameters) {---your code here}
  + **THIS IS OBJECT CONSTRUCTOR** 🡪 Now you can add it w properties and labels
    - * var myObj = new Object();
  + This tells JavaScript: "I want you to make me a new data type, and I want that data type to be an Object.
  + You can add keys to your object after you've created it in two ways:
    - * myObj["name"] = "Charlie";
      * myObj.name = "Charlie";
      * Note that I never had to initialize the name of the internal data, nor did I had to initialize it’s data type.
  + You can initialize other objects instead of an object like this
    - * var mainObj = {
      * obj1: {},
      * Obj2: {}
      * };
      * Or like this
      * mainObj[obj1] = {};
      * mainObj.obj2 = {};
      * or ----------------------
      * var mainObj = new Object();

mainObj.obj1 = new Object();

* + - * mainObj.obj2 = new Object();
  + The keys of an Object in JS are called properties or category label (e.g. obj.name)
    - **dot notation**. So to access a property, we use ObjectName.PropertyName(e.g., bob.name)
    - **bracket notation**. In this case we use ObjectName["PropertyName"] to access the desired property. Note, we need " " around the property's name.
* **Methods** are just like functions, but associated with an object.
  + We call a method like a function, but we useO bjectName.methodName().
  + E.g setAge is a method of the object bob
    - * ****
  + Methods can be used to change object property values. They can be used to make calculations based on object properties. Functions can only use parameters as an input, but methods can make calculations with object properties.
  + We can make a method work for many objects using the keyword, **this**. this acts as a placeholder, and will **refer to whichever object called that method** when the method is actually used.
    - E.g
      * ****
  + This can also be bounded to one class
  + this is still a placeholder, but in this scenario, this can only ever refer to rectanglebecause we defined setHeight to be explicitly part of rectangle by defining it asrectangle.setHeight.
  + E.g.
    - * 
  + To make Objects more generic, allowing you to pass in parameters in its constructor, you would do it this way:
    - * ****
  + Prettier way to declare a method as a property of a Object
    - Var objName = { methodName: function(parameters){--- code here--} };
    - E.g.
      * + ****
* **Prototype**
  + if you want to add a method to a class such that all members of the class can use it, we use the following syntax to extend the prototype:
    - className.prototype.newMethod =
    - function() {
    - statements;
    - };
    - Better example
    - ****
* To declare **private variables** within a class, instead of using this in the constructor, use var.
  + The Person class has been modified to have a private variable called bankBalance. Notice that it looks just like a normal variable, but it is defined inside the constructor for Person without using this, but instead using var. This makes bankBalance a private variable.
  + E.g:
    - ****
    - User getters to access private information
* To declare **private methods** within a class, use *var* instead of this once declared
  + Changing this.returnBalance from the last example to var returnBalance makes this method private. If you run the program trying to access the method you get an undefined error this time.
    - ****

**Comparison operators**

* Equal to is also === in JS
* Not equal to is also !== in JS
* On JS UNLIKE Java, the === operator compares booleans instead of doing (!true); this can also be used to compare strings and everything else
  + The identity (===) operator behaves identically to the equality (==) operator except no type conversion is done, and the types must be the same to be considered equal.
  + The == operator will compare for equality after doing any necessary type conversions. The ===operator will **not** do the conversion, so if two values are not the same type === will simply return false. It's this case where === will be faster, and may return a different result than ==. In all other cases performance will be the same.
  + The == and != do the right thing when the operands are of the same type, but if they are of different types, they attempt to coerce the values. the rules by which they do that are complicated and unmemorable. These are some of the interesting cases:
    - * '' == '0' // false
      * 0 == '' // true
      * 0 == '0' // true
      * false == 'false' // false
      * false == '0' // true
      * false == undefined // false
      * false == null // false
      * null == undefined // true
      * ' \t\r\n ' == 0 // true
  + The lack of transitivity is alarming. My advice is to never use the evil twins. Instead, always use ===and !==. All of the comparisons just shown produce false with the === operator