Language Fundamental

JavaScript

**JavaScript Under The Hood**

* **Conceptual Aside** 
  + ***Syntax Parser***, is a program that read/compiles your code and convert it to something the computer can understand
  + ***Lexical Environment***, is where something sits physically in the code you write
  + ***Execution Context***, is the wrapper that manages code that is running it holds ***lexical environments*** as well other things
  + ***Name/Value Pair***, is a name which maps to a unique value. In ***Execution context*** the name can be defined more than once but can only have one value. An example would be a variable set to value ex. address = “100 main St” the address is the ***name*** & the string is ***value***
  + ***Objects***, are collections of name value pairs. When talking about ***objects*** in javascript we’re talking about a collections of name value pairs that ***value*** can also be another ***object***.
* **Global Environment & Global Object**
  + When your code runs javascript engine the ***Execution context*** creates 2 things for you ***Global object*** & ***‘this’***
  + When running javascript in the browser the ***global object*** is the ***window.***
  + On the global level, the ***global object*** which is the ***window*** is equal to ***When your code runs***.
  + ***Global*** in javascript simple means not inside of a function, so code or variable that are not inside of function are ***global***
  + When you create ***variable*** or ***function*** and you’re not inside of function those ***variable*** & ***functions*** get attached to the global ***object***
  + So when your code is runs a ***Execution Context*** is created which creates a ***Global Object(i.e Window)*** & special variable called ***‘this’*** it also holds your code if you’re are writing ***global code*** which is simply code outside of a ***function***
  + This is all being done for you by the javascript engine
* ***Creation & Hoisting in Execution Context*** 
  + When your code runs javascript engine the ***Execution context*** creates 2 things for you ***Global object*** & ***‘this’***
  + ***Execution Context*** is created in 2 phasesthe first is the ***creation phase***
  + The ***creation phase*** is where the ***Global Object***, ***‘this’***, ***Outer Environment*** & also in the phase it setup memory space for variable & functions this is where ***Hoisting*** occurs
  + ***Hoisting*** doesn’t actually move code to top the page, before your code run javascript has already set aside memory space for the ***variables*** you created in the entire code so those ***function*** & ***variable*** exist in memory, When the code execute line by line it can access them.
  + However when it’s setting up ***variables*** the ***variable*** gets created but is set to ***undefined*** in the creation phase but get assigned in the ***execution phase***
  + All ***variable*** in javascript are set to ***undefined*** & ***functions*** are sitting in memory in there entirety
  + ***Undefined*** is a special value that javascript has it within it that means that the variable hasn’t been set.
  + The second phase in the ***Execution Context*** is called ***Execution phase*** this is the phase where it runs your code line by line.
* ***Function Invocation and the Execution Stack***
  + ***Single threaded***, one command at a time
  + ***Synchronous,*** one at a time & in order
  + Javascript is ***Single threaded Synchronous*** execution which means one thing is happening at a time.
  + Every time a function is ***invoked*** it gets placed on top of the ***global Execution Context*** which creates the ***execution stack***
  + ***Execution stack*** runs from top to bottom one by one & every time a function is ***invoke*** it gets placed on top of the ***execution stack*** & which everyones on top is the one currently running
  + Every time you ***invoke*** a function it creates a new ***execution context***
  + The ***execution context*** that has been created goes through the same process as the global one.
  + It will have its own space ***variables*** & ***functions*** it will go through the ***create phase*** then it will go execute line by line inside of the ***function***
  + If there is another ***function invoked*** inside of that ***function*** its going to stop on that line of code then create another ***execution context*** which get added to stop of the ***execution stack***
  + This is how ***function invocation*** happens in javascript, every function creates new ***execution context,*** which goes through the create phase then excuses the code line by line.
  + ***Lexical*** the order doesn’t the order of which you function are placed doesn’t matter
  + Every time a function is ***invoked*** a new ***execution context*** is created for that function, the ***‘this’*** variable is also created for that function, the variables are set up during the ***creation phase*** than the code is executed line by line
  + When the ***function*** is getting executed line by line if there’s a function being ***invoked*** inside of it the current function stops then the ***new function*** gets add to the top of the ***stack*** then that function
* ***Scope***
  + ***Variable Environments,*** is where the variable lives & how they relate to each other in memory
  + ***Functions*** have there own ***execution context*** so ***variable*** that are changed inside of the ***function*** don’t affect what’s outside of the ***function***
  + When a ***function*** has ***variable*** inside of it’s value it reference to its outer environment
  + The ***outer environment*** that’s outside of function is called ***lexical environment***
  + So every variable that’s declared outside of function has access to inside of the functions context
  + The chain that that gives you access to the ***outer environment*** is called the ***Scope chain***
  + When a function is using a variable it looks for the value of that variable outside of its ***lexical environment*** and it keep going outside of it ***lexical environment*** till it finds it if it doesn’t find it it return as ***undefined***.
  + Where a function sits determines it ***outer environment*** ***reference***, but as its being executed those ***execution context*** are stacking up & its running ***synchronously***.
  + ***Execution context***, ***execution environment***, ***variable environment***, ***lexical environment*** all of these thing ultimately are defining what called ***scope***
  + ***Scope*** is where a variable is available in your code & if its truly the same variable or a new copy.
  + ***Let*** allows the javascript engine to use ***block scoping*** during the ***execution phase*** the variable still get placed into memory & set to undefined but you’re not allowed to use it till that line of code is ran in the ***execution phase*** which actually declares the variable
  + ***Let*** are declared inside of a block & only available in that block
  + ***Blocks*** are anything inside of curly brackets
* ***Asynchronous Callback***
  + The ***outer environment*** that’s outside of function is called ***lexical environment***
  + In the browser there’s other things that run with the javascript engine I.E ***Rendering engine*** & ***HTTP Request***
  + While javascript is ***Synchronous*** there are other things that runs with the javascript engine like the ***rendering engine*** & ***HTTP Request***
  + ***Event queue,*** periodically runs after the executions stack has finished
  + It looks to see if a particular function should be run when the event was triggered and create a ***execution context*** for that function
  + So javascript isn’t ***asynchronous*** but the browser is putting things in the event queue but the code is still running line by line
  + The ***javascript engine*** doesn’t look at the ***event queue*** till the stack is empty & after the stack is empty it consistently looks for events & process those event in the order they happened
  + ***Asynchronous*** are possible in javascript but the ***Asynchronous*** part is what’s happening outside of the ***javascript engine***

***Types & Operators***

* **Conceptual Aside** 
  + ***Dynamic Typing***, means you don’t tell the engine what type of data a variable holds it figures it out while your code is running
  + ***Variables*** can hold different types of values because its all figured out during execution
  + ***Javascript engine*** determine what type you’re giving the variable
* **Primitive Types**
  + ***Primitive type***, is a type of data represents a single value so something that isn’t a object
  + There are 6 ***Primitive types*** in javascript

1. ***Undefined***, represents lack of existence this you should set undefined to a variable.

2. [Null](http://www.apple.com), also represents lack of existence but the different is you can set a variable to this

3. ***Boolean***, is True or False

4. ***Number***, is a floating point number meaning there’s always some decimal there’s only one number type in javascript and it can make math weird

5. ***String***, a sequence of characters

6. ***Symbol***, This primitive type is useful for so-called "private" and/or "unique" keys.

* **Operator**
  + ***Operator***, are a special function that are written differently generally operators take two parameters & return one result
  + An example of a operator is ***plus(+)*** or ***minus(-)*** these are simply functions and javascript gives us the ability to use them and this is called ***Infix notation***
  + All these ***operators*** are essentially a function call that take in 2 parameters and returns a value
  + Its important to understand what’s happening instead of these function
* **Operator precedence & associativity**
  + ***Operator precedence***, are which operator function gets called first functions are called in order of precedence higher precedence wins
  + ***Operator associativity***, are simply what order an operator function gets called in either left to right or right to left
  + When a operator has the same ***precedence*** it then uses ***associativity***
  + Link to ***precedence*** order***,*** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>
  + When multiply ***operator*** are being used the one with the higher ***precedence*** runs first this is important to note for finding bugs
  + When is has the same ***precedence*** then it uses ***associativity*** to determine the result
  + ***Parentheses*** ( ) is called grouping and this is has the highest precedence so when something gets grouped together it will always run first
  + ***Precedence*** & ***associativity*** are really important because javascript it dynamically typed
* **Coercion**
  + ***Coercion***, Converting a value from one type to another this happens often in javascript because its dynamically typed
  + An example of ***coercion*** would be 1 + “1” would result in 12 because javascript coerced and determined the value
  + Since ***operators*** are just functions ***coercion*** is happening inside of the operator functions
  + ***coercion*** is a fundamental part of the javascript language
* **Comparison Operators**
  + ***Comparison Operators***, are operators that compare values i.e >(greater than) <(less than)
  + Its not always obvious a particular type is going to ***coerce*** to, this can cause bugs and odd problems if you don’t understand what’s going on
  + An example is ***undefined*** returns ***NaN*** while ***Null*** would return 0
  + A way to not ***coerce*** when checking if 2 things are equal is using the == (***equality***) checks if 2 things are equal without ***coercing***
  + == (***equality***) can cause strange errors and doesn’t properly check return values because of the unexpected ways it behaves
  + === (***Strict equality***) compares two things but doesn’t ***coerce*** the values at all it simply doesn’t coerce the values.
  + When === (***Strict equality***) operator checks the types & if the types aren’t the same you’re going to get false
  + When making equality compares 99% of the time you will use === (***Strict equality***)
* **Default values** 
  + You can set default values using the ||(***or***) operator
  + ||(***or***) operator doesn’t just return true or false, it returns values that can be ***coerced*** to true
  + Using the ||(***or***) operator can prevent you from needing to use if statements and make your code clear
  + The ||(***or***) is commonly used in libraries to check if there’s global value so it doesn’t conflict with another library that might be on top

***Object & Functions***

* **Object & the Dot** 
  + ***Object*** are collection of name/values pairs that can have nested name/value pairs
  + An ***object*** can have ***properties*** & ***methods***, it can have ***Primitive*** **properties** such as boolean, string etc..
  + ***Objects*** can also have another ***object*** nested inside of it which also considered a ***property***.
  + **Objects** can also contain function & when a function is inside of object that called a ***method***
  + These objects sit in memory & it will have references to these different ***properties*** & ***method*** which are also in your computers memory
  + [](***Computer Member Access***) takes properties name looks for it inside of an object
  + [](***Computer Member Access***) can be used to find properties inside of an object and return a value
  + Another ***operator*** is ***Dot*** operator since its an operator just like [] it takes 2 parameters, Dot has a higher ***precedence.***
  + *The* ***Dot*** takes 2 parameters which are the ***object*** you’re looking and the name of the ***property***
  + ***Object Literals***, is syntaxes that uses curly braces to define name/value pairs
* **Faking Namespaces** 
  + ***Namespace,*** is a container for variables & functions its used to keep variable & functions with the same name separate
  + The reason its called “faking” is because javascript doesn’t actually have ***Namespace*** but its doable
  + The idea is to contain the ***properties*** inside of container object so it doesn’t have conflicting names, this is commonly done in ***frameworks***
* **JSON & Object** 
  + ***JSON,*** stands for JavaScript Object Notation it looks a lot like ***object literal*** syntaxes but its not the same thing
  + ***JSON*** is used to send data all over the internet & was inspired by ***object literal***
  + The difference is ***Properties*** need to be wrapped in quotes
  + Anything that is ***JSON*** valid is also ***Object Literal*** valid but not all ***Object Literal*** are valid ***JSON***
  + ***JSON.stringify()*** convertes Object literall to JSON & ***JSON.parse()*** convertes JSON to object literall
* **Functions Are Objects**
  + ***First Class Functions,*** everything you can do with other types like object, string, boolean etc.. you can do with functions like assign them to variable, pass them around & create them on the fly.
  + ***Functions*** are a special type of ***objects*** the have all the normal features as normal ***object*** with some other special properties
  + You can attach ***properties*** & ***methods*** to a function because it just an object
  + Functions have 2 properties ***Name*** which is optional & the ***CODE*** the code you write get places into a special property in the function object
  + The code you write isn’t the function, the function is ***object*** with other properties & the code you write is one of those other properties you’re writing on to it
  + What makes the ***code*** property special is that its ***invocable***() which runs the code sitting inside of that property & that’s when the ***execution context creation*** & ***execution*** occurs
  + It's fundament to understand that ***functions*** are just an ***object*** with other ***properties*** to it & it has the same abilities as object such as being moved around, copied & given to other elements or other areas of your code.
  + You can attached any ***property*** like ***string*** you want to a function & then use ***Dot*** to find that value
  + ***Functions*** are more than just container of code it's an object, and as such you can pass it around, it sits in memory at a specific location. it has properties, methods BECAUSE FUNCTIONS ARE OBJECTS!
* **Function Statements & Function Expressions** 
  + ***Expression,*** is a unit of code that results in a value it doesn’t have to save to variable i.e a = 3, 1+2 these are ***expressions*** because the result a value
  + ***Statements***, don’t return a value and simple run such as if’s statements
  + ***Statements*** just do work & ***Expression*** result in a value
  + Since functions are objects in javascript we have both ***function statements*** & ***function expressions***
  + ***Function statements*** are functions that aren’t stored in variable & can be invoked anywhere.
  + ***Function expression*** are functions that are stored in a variable and can only be called below it because since its stored in variable it initial set to undefined.
  + You can pass a function to a function to call the function that was pass through you can simple ***invoke***() it
  + The concept of first class functions where you can pass functions around, give functions to other functions, use them like you do variables introduces gives us the ability to do ***functional programming***
* **By value VS By reference** 
  + ***By value,*** when a ***primitive values*** like string, boolean …etc, gets set to a variable it gets a location where it sits in the memory when a new variable or function get set to the first variable if its ***primitive value*** the new variable points to a new location in memory a copy of that value gets placed in memory
  + When passing or referencing or setting equal one ***primitive*** ***value*** to another by copying the value is called ***by value***
  + ***By reference,*** when an ***object*** get set to variable then that variable is set to new variable instead of getting a new location in memory is simply points same location in memory a does.
  + ***By values*** is when a primitive value gets set to a new location it creates a copy of that primitive value, ***By reference*** applies to that objects instead of creating a copy of that **object** it points it to the original **object**
  + When working with ***object*** because of ***by reference*** when you mutate that variable anywhere the original changes because there point to same location in memory this also applies to ***function*** since ***functions*** are ***object***
  + Even when ***objects***(***by reference***) or ***primitive***(***by value***) gets passed through a function as ***parameters*** the same concepts get applied
  + ***Equals operator*** sets up new memory space so when you set a variable using ***Equals operator*** to a pre-existing variable it will overwrite that variable because ***Equals operator*** doesn’t recognize it exists
  + All ***primitive*** ***type*** are ***by value*** & ***object*** are ***by reference***
* **Objects, Functions, & ‘This’**
  + In the ***creation phase*** of the ***execution context*** along with ***variable environment*** which is where the variable created inside of that function live & ***Outer environment*** where it sits physical in the code which tell it how to look down the scope chain so if you ask for a variable & the variable isn’t there go out further to look for it also includes the ***“this”*** variable
  + ***‘this’,*** points at a different object depending on how the function is invoked
  + There a few scenarios where ***‘this’*** will be changed depending on how the function is called
  + Whenever a function wether its ***function expression*** or a ***function statement*** the ***‘this’*** variable will point to the global object even if there are more then 1 ***execution context***
  + When you use the ***‘this’*** variable inside of a function and add a property to it you’re actually added to global namespace but this just when you’re ***()invoking*** the function ***‘this’*** points to the global variable
  + When the ***‘this’*** variable is used inside of a ***method*** which is a function inside of ***object*** it will point to the ***object***
  + When you have a nested function inside of a ***method*** and use the ***‘this’*** variable it doesn’t point to the parent ***object*** but the global ***object*** even if its inside the ***object*** you created.
  + Creating a variable that stores the ***‘this’*** and referencing that variable instead of the ***‘this’*** is a good pattern to make it reference the proper parent ***objects***
* **Arrays** 
  + ***Arrays***, Are a collection of anything that start with a 0 index
  + Since javascript is ***dynamical typed*** it figures out the types of things on the fly so you can mix & match each individual item an ***array*** that are different types
  + ***Number***, ***boolean***, ***strings***, even ***objects*** & functions can be stored inside of an array.
  + When a function is inside of an ***array*** its a ***function expression*** because javascript understand that this is creating function on the fly there so its not a ***function statement***
  + You can even ***invoke*** function inside of ***array*** by finding it location using the [](***literally syntax***) and its index then ()***invoke*** the function
* **Arguments & Spread**
  + ***Arguments,*** is simply just another name for ***parameters*** you pass to a functions
  + The keyword ***Argument*** is another keyword like ***‘this’*** javascript set-up for you automatically when you execute a function & it hold a list of all the values of all the parameters that pass to a function.
  + ***Spread***, is the new approach of what ***argument*** does currently
  + When you create ***arguments*** in javascript takes care of those parameters for you because of hoisting where it sets up the values in the memory space so those ***arguments*** when not set to a value are default ***undefined***
  + When values are passed to ()***invoking*** function they get defined from left to right
  + When you console.log ***argument*** keyword it return a list of all the values of the ***parameters*** that were passed in array like structor but *[]*italicized without all the features of an array
  + ***Argument*** are deprecated we use the …***spread*** operator which takes all the parameters and wraps it up in javascript ***array***
* **Asides & refresher**
  + the code we write isn’t directly run on our computer it between the code we write and the computer there’s the program that translate your code into something the computer can understand in the browser its the ***javascript engine*** & it has different aspect to it and one of them is the ***syntax parser***
  + ***syntax parser*** reads your code & determines if its valid & what is you’re trying to do, goes character by character and keeps on going and in knows what you intend to do
  + ***syntax parser*** makes assumptions, stating certain rules and can even make changes to your code before its even ***executed***
  + You should but your own ***semicolon*** because you don’t want javascript engine to make the decision for you, you want to be certain that are writing the code as it should be.
  + Its really important to put a ***semicolon*** after your ***return*** statement because if the code your ***return*** isn’t on the same line as your ***return*** statement it put a ***semicolon*** in front of the ***return*** statement.
  + Always make sure your ***return*** ***keyword*** is on the same line as your ***return statement***
  + ***Whitespace,*** is invisible characters that create literal ‘space’ in your written code which is created by the ***syntax parser***
* **Immediately Invoked Function Expressions(IIFE)s**
  + ***Function statements*** is new statement where function is the first word & javascript store it in memory but you have to ()***invoke*** it to execute but can be ()***invoke*** anywhere
  + ***Function Expressions*** are functions that are stored in a variable & isn’t put into memory initially but rather during execution when javascript finds the code in the execution phase so it need to be ()***invoke*** after the function is written
  + ***IIFE***, are simply when you immediately ()***invoke*** a ***function expression*** by placing () after the {}***brackets*** it lets you run at the point you created
  + So when ***‘function’*** isn’t the first word it sees then the ***syntax parser*** throws an error so when you use ()***grouping*** operator to create a ***function expression &*** it will immediately ()***invoke*** that function
  + When a ***IIFE*** is ran in javascript the first thing that is created is the ***Global Execution context*** in the ***global context*** in the ***creation phase*** when it sees the ***function expression*** part it creates a function ***object*** in memory & since its ***Anonymous function*** it just ***object*** that has code in it.
  + Then is sees ()***invokes*** operator which creates a new ***Execution Context*** so the variable being passed through goes into that functions ***execution context variable environment***.
  + So since the variable goes into its own ***execution context variable*** ***environment*** & not to the ***global*** that why this makes it such a useful approach.
* **Understanding Closures**
  + In a function you can ***return*** a function since functions are object so you can ***return*** it as value
  + When a function return is returned inside of function only 1 ***execution context*** gets created and pop off the stack
  + When a ***execution context*** gets pop off stack so since every ***execution context*** has a space in memory even when get pop off the stack the memory stays
  + So when there’s a function that returns a value stored in the memory another function that’s invoked after that ***executions context*** has reference to memory of its outer space
  + So when a ***execution context*** grabs all the variables its suppose to have access to is called ***closure***
  + ***Closure*** aren’t suppose you type or tell the javascript engine to do closures are just simply a feature of the javascript that just happened
  + It doesn’t matter when we invoke a function, we don’t have to worry if the outer environment is still running javascript will always make sure whatever function I'm running it will make sure it will have access to the variables that its supposed to have access to.
  + When javascript closes in on all the variable is suppose to have access to is called ***closure***
  + When a function is being executed later it only going to be able to tell you what is in memory right now not in the time we created the function
  + Using an ***immediately invoked*** function which would great a new execution context for each function inside of loop and return us the expected output
  + Closure are used in frameworks to create encapsulate functions that have different values returned.
  + When you return a function inside of function the returned function has access to what the passed variable was at the time it was created by pointing to that memory space and that’s how you create function from other functions in javascript because functions are objects.
* **Callback Functions** 
  + ***Callback Functions,*** are functions you give to another function to be run when the other functions is finished
  + The function you call/invoke, “***calls back***” by calling the function you give it when it finishes
* **Call(), Apply(), & Bind()** 
  + ***call(), apply(), & bind()*** are used to control what the ***‘this’*** variable ends up being when the ***execution context***is created
  + All functions objects in javascript get access to these 3 functions
  + ***Bind()***, returns a new functions it create a copy of the function its applied too & sets up a new function so whenever it’s runs it sets up ***‘this’*** variable of whatever is being passed through it.
  + So whatever value is passed through the ***bind()*** function because the new ***‘this’*** variable
  + ***Call()***, also allows you to determine what ***‘this’*** variable is and it immediately invokes that function similar to using () but instead you can determine what ***‘this’*** is
  + The different between ***bind()*** & ***call()*** is bind creates a copy of the function and lets you determine the ‘***this’.*** ***call()*** invokes it and doesn’t make a copy but still lets you determine the ***‘this’***
  + ***Bind(),*** doesn’t execute the function is create a copy and executes that function
  + ***Apply()***, does the same exact thing ***call()*** but takes an ***array*** instead of normal list
  + All these three functions takes in 2 ***parameters*** the first one letting it know what ***‘this’*** is the other one the value you’re passing in
  + These functions can be used to grab ***method*** from other ***objects*** and use them
  + ***Bind()***, can be used to ***curry*** function where you can create a copy of a function and give it a permanent ***parameter*** value
  + Function ***currying***, creates a copy of a function but with some preset parameters to it.
* **Functional Programming**
  + Creating a function that gets passed to some other function that accepts a function that does work is the concept of ***Functional Programming***
  + The think process of ***functional programing*** is start thinking of how can you give your functions, functions or return function from your functions in order simplify the code you’re writing over & over again
  + ***First-Class*** functions are functions that are treated like any other variable, in javascript since functions are ***objects*** functions are ***first class objects***
  + The ability to pass functions as ***argument*** to other functions, can be returned by another function and also can be assigned as a value to ***variable*** is what gives us the ability to use ***functional programing*** in javascript
* **Object-Oriented Javascript & Prototypal Inheritance**
  + ***Inheritance***, is one object gets access to the properties & method of another object
  + ***Classical inheritance***, is what the most popular way of using inheritance and used in most popular languages
  + ***Prototypal inheritance***, is simple flexible, extensible & easy to understand and its what ***javascript*** uses
  + All ***objects*** in javascript has a ***prototype{}*** property and since functions are ***objects*** that includes functions
  + Property or ***prop*** is simply a reference to another object
  + ***Prototype Chain,*** looks for where we have access to property or method amongst a sequence of objects and there connect by ***prototype chain***
  + If there is another object that other object can point to the same object as its ***prototype*** so objects can share all ***prototypes*** if they wanted to
  + When ***javascript engine*** goes down the ***prototype chain*** they happen to be pointing at the same place which allows objects to share the same ***prototypes***
  + When using ***prototype*** to look for a ***property***, the ***‘this’*** variable is referred to the original object because the ***javascript engine*** knows where it originated from
  + The ***javascript engine*** starts and top of the chain and works it way down & stops the first time it finds what its looking for
  + The ***‘this’*** variable references the original object when the original object isn’t available it looks at the ***prototype*** it referencing and find its there
  + In ***javascript*** everything is either an ***object*** or a ***primitive*** everything that isn’t a ***primitive*** in ***javascript*** they all have ***prototype*** expect the ***base object***
  + Everything always eventually leads the ***base object*** which is at the bottom of the chain
  + The reason these ***methods*** are available is because the javascript engine sets the ***prototype*** for you and used the prototype chain to find those ***method*** & ***properties*** in base object
  + All ***arrays***, ***functions***, & ***objects*** have ***prototype*** pointing to an object that that has those function method available for each of those ***non-primitive*** data types
  + The ***prototype*** of ***prototype*** is the ***base object*** which is the bottom of the prototype chain always so that means ***arrays***, ***functions*** & ***objects*** all come from a ***base object*** and the ***javascript engine*** creates those ***data-types*** with those methods available to us
  + So because everything eventually leads to ***base object*** thats NOT a ***primitive*** type like ***Boolean***, ***string***, ***number***, ***undefined*** & ***null*** everything else like ***functions***, ***object***, ***arrays*** is consider a object because it leads to the base object in the ***prototype chain***
  + That is what meant when people say everything is ***object*** in javascript
* **Reflection & Extends**
  + Another aspect of creating objects is called ***extend*** & ***extend*** is possible because of something called ***reflection***
  + ***Reflection***, an object can look at itself, listing & changing its ***properties*** & ***methods***
  + ***Extend*** is the concept is of grouping together an object to contain all the property an methods of the originals
* **Function Constructors**
  + There are different ways to create objects in javascript one was is using the ***‘new’*** keyword this method was created to attract java developer
  + ***‘new’*** is a operator it immediately creates an empty object, when you use ***‘new’*** keyword its changes what ***‘this’*** variables points to it points to empty object the ***‘new’*** operator created
  + ***Function Constructors***, is a normal function that is used to construct objects, the ‘this’ variable points a new empty object and that object is returned from the function automatically
  + Any time you create a function object, all functions has a ***prototype*** which starts off its life as an empty object
  + This ***prototype*** that given to all functions are only used by the ***‘new’*** operator this prototypes is only used for when your using a function as ***function constructor***
  + All functions in javascript gets the ***prototype*** property, when you call the ‘new’ keyword which create an empty object and it sets the ***prototype*** of that empty object to prototype property of the ***function constructor*** you then call
  + When creating objects its a good idea to set properties inside of a ***function constructor*** but ***method*** can sit on the prototype since functions in javascript are objects they take up memory space so when you place a function inside of ***functions******constructor*** it will create a copy of that method and take up more memory space but if you added it to ***prototype*** you only have it once and can be reused and wouldn’t take up memory space
  + So you for efficiently its better to add a method to ***prototype*** because you only need 1 copy to be used
  + Any function we intend to be a ***function constructor*** we use a capital letter
  + ***Function constructors*** are the old way of creating object and that all its used for to create objects
  + ***Array*** are just objects with additional properties and there key value pairs are different so you can’t use for..in loop for ***arrays***
  + ***Arrays*** are objects and using for..in you can iterate down into the prototype which is not safe, items are added properties
* **Object.create & Pure Prototypal Inheritance**
  + ***Object.create*** is another way to make objects in javascript, this is javascript unique way to create objects not like ***functions constructors*** which was included to resemble other language
  + ***Object.create***, creates an empty object & the prototype points to whatever you pass in to ***object.create()***
  + The pattern is to override whatever you want to by adding the properties to the created object
  + You can keep methods on the base object, you can add new method & properties to that base object and all your created objects using ***object.create*** will suddenly get them
  + This concept of using ***object.create*** and following the pattern is called ***Pure Prototypal Inheritance***
* ***Classes & ES6***
  + The newest way of creating objects in javascript is using classes, Its just another way of creating object in javascript
  + Even though javascript has a ***class*** keyword its still just an object in javascript, you’re just creating an new object from that a object
  + ***Extends***, is used to set prototype(\_\_proto\_\_) value in classes
* ***ODDS & ENDS*** 
  + ***Typeof*** of keyword tells us what type thing is this, it can used on primitive datatypes and objects
  + ***Typeof*** works a little strange on array because in javascript array are objects so it returns object instead of array
  + ***InstanceOf***, tells you if any objects down the prototype chain finds that type of object you’re using it on
  + When you use ***typeof*** on ***null*** you get returned a object this bug in javascript
  + ***Strict Mode***, lets you tell the javascript engine to be more strict and it can help you prevent errors in certain situations