# Scope

* + Interpreter scans all the code before it begins to execute
  + First pass compiles the code and second pass executes the code
    - Scope of variable declarations
    - Function compilation may be deferred if it has no ‘call’
    - Function may be recompiled (Hot Swap) based on how its being used (parameters being passed and datatypes etc.)
    - Implicit declaration for ‘parameters’ parameters have local/function scope
  + Variables are references with a certain scope.
  + Undeclared
    - When a LHS variable/reference is not declared it’s created at runtime in Global Scope.
    - For RHS a reference error will occur for undeclared functions/references
  + “Undefined” means uninitialized or no value assigned. Whereas “undeclared” gives reference error
  + Re-declaration
    - Re declaration of variables are ignored/have not effect. However, re-initialization could happen.
    - In case of functions, the latest declaration overwrites previous declaration
  + Named Functions and anonymous functions
    - Function declaration: Registered in the scope(using function name) and can be referenced elsewhere within same scope
    - Function expression: Scope is limited to itself
      1. Anonymous. Cannot be referenced from Self. Difficult to debug.
      2. Named: Can be referenced from within function itself
    - IFFE
      1. Can be used to create “hidden” function.
         * (function () {} ) (); //immediately invoked function expression. IIfe
         * Code will only be executed once.
         * Hide/Wrap code in its own scope. Only {} will not work
      2. Can be “named” or “anonymous”. Named will be help in debugging.
      3. Allows to pass variables (global) as an alias in an enclosing scope. This avoids conflicts
      4. Whole code can be IFFE so that there are no conflicts
  + Block Scope e.g. catch block, {let }, with
  + Lexical Scope(author time) vs Dynamic(runtime) Scope
    - Scopes cannot overlap. They are either fully nested or exclusive. At compile time all references are identified.
    - Dynamic scope
      1. Using declarations in ‘eval’. : Modifies lexical scope at runtime
         * Avoid ‘eval’ or string in setTimeout and setInterval. Can
      2. With : short hand for objects. Avoid using it. Creates “new” lexical scope at runtime
  + ES6 Block scope **‘let’** – limits scope to the block its defined only instead of function or global
    - However, has to be at **the top of the block** for it to be available for the entire block.
    - Block Scope can change when you refactor and add braces. Variable could suddenly become out of scope. Hence **look to create explicit block scope** instead of implicit(let within braces) block scope
  + **Hoisting**:
    - All declarations are processed at compile time.
    - Function declarations are hoisted before variable declarations
    - Expressions will be processed only at runtime. Hence function expressions are not ‘hoisted’.
    - ‘Let’ does not hoist and hence has to be at the top of the block
  + Execution Context: this
    - Behavior depends on the context(how and where) the function is invoked. Crossbridge is not possible

1. Was it called with the new keyword
2. Call or Apply(bind, arguments). For predictability of ‘this’ object
   1. foo(‘bam’) 🡪 Hard Binding (predictable) : foo = foo.bind(obj). .apply. Too much use of hard binding means that you are better of using the lexical scope
   2. foo.call(obj)🡪 Explicit : Use ‘obj’ for ‘this’
3. O2.foo() 🡪 Implicit: This = O2
4. foo() 🡪 Default: Strict mode then ‘this’ = undefined else this = global object

## New Keyword

* + Can be used to invoke any ‘function’. Function is called as if it’s a constructor
  1. A ‘new’ object is created when you use ‘new’ keyword. A function is converted to an object
  2. New object is “linked” to a ‘different’ object
  3. ‘new’ object becomes this for the ‘function’
  4. Calls constructor
  5. If there is no ‘return’ in the function then implicitly ‘this’ will be returned

## Closure

* + Capability of a function to access its “lexical” scope even when the function is executed outside that lexical scope
  + Classic “Module” Pattern
    - Must have an outer wrapping function to be executed
    - One or more functions must be returned from this function. The inner function will have access to the private member of the function. Public API
    - Wrap you code into IFFEs and make them into modules
  + Modern “Module” Pattern – used by ‘Require’
  + ES6 module pattern: file acts as wrapping function. “export” acts as returning function reference as a public API
  + Module Patterns promote abstraction
  + Var publicAPI = {init : init; loadData : loadData } return publicAPI
  + Handling of external references

## Javascript Classes/Object Orienting

#### Reference Types

* + Reference Types are the closest things to “Classes”
  + **Constructor**: A function with **the same name** as the class/reference type
    - The constructor can be used with the new() to create an ‘instance’ of the reference type
  + **Other methods** can be defined as follows
    - classname.**prototype**.methodname = function() {}
  + **Properties:** Define using the ‘this’ keyword
  + **Animal**
    - function Animal(name) {} // Constructor
    - Animal.prototype.sayName = function {} //Other method
    - Dog.prototype = new Animal() //define inheritance
    - Dog.prototype.constructor = Cat //???
    - function Dog(name) { Animal.call(name) } //Child constructor calls Parent Constructor
    - Dog.prototype.sayName = function() {} //Extend Method



* There exists a Global function(‘Object’). It is **“linked”** to a Global Object via “Object.protoype”
* Global Object contains toString, valueof etc
* When a function(“Foo”) is declared, a new Object will be created in memory. Only 1 copy of this function
* Foo “function” is linked to this new Object via Foo.prototype. The new object has a link “.constructor” pointing to Foo Function. New object is also linked to Global Object via “\_proto”
* Foo.prototype.identify is a “property” in the new Object.
  + **New**
    - A constructor (invoked using “new”) makes an object “linked” to its own prototype
    - new Foo(“a1”) creates a brand new Object (a1) linked[[p]] to Object(Foo.prototype).The “this” context gets set to the new object. A property is created by the constructor in the new object. ‘this’ is returned by the constructor.
    - Each object created with new can have properties that don’t exist/defined in the prototype e.g. a1.speak and a2.bark
    - Constructor property exists only in the prototype. Prototype scope wraps the new object scope
    - a1.\_\_proto == this.[[p]] == a1.[[p]] == Foo.prototype == Object.getPrototypeOf(a1) = a2.constructor.prototype
    - a1.identify = //add new property (overridden) to a1
      1. Foo.prototype.identify.call(this) -🡪 super.identify();
      2. Avoid this by giving different method names in parent and child. Avoid “shadowing”

a1[“identify”] = “”; //can be used to add “dynamic” properties

* + - LHS uses lexical scope
    - RHS. ‘This’ tell us which building to go based on the call site. Prototype helps us how to find properties if it does not exist in the direct object . Traverse up the prototype chain. Object.prototype is analog to the lexical scope.
    - Lexical scope and protype scope does not overlap
    - Object.create(Foo.prototype) (Step1 and 2) , constructor is lost
  + Object.freeze
  + Object.defineProperty writtable: false, enumerable, configurable?. get,set property
  + Object.Create will not call constructor. New will execute Animal function
  + **EC6**
    - class Animal { //Explicit class definition

**constructor (name)** {

this.name = name;

}

sayName() {

console.log (this.name);

}

}

* + - class Dog **extends** Animal { //Explicit Inheritance

**constructor (name)** {

**super**(name); //Explicit Super

}

sayName() {

console.log (this.name);

}

bark() {

console.log(“woof”);

}

}

* Linking as opposed to inheritance 🡪 “Behavior Delegation”
* OLOO 🡪 objects linked to the other objects
  + b1 = Object.create(Bar)
  + Bar = Object.create(Foo)
  + Var Foo = {}
  + Use unique function names as opposed to polymorphism and delegate behavior from one object to another
  + Object delegate behavior to its prototype
  + No copies of the function are created

95% time use module pattern and 5% OLOO pattern

Think about “utility” object instead of “parent”

* Write init() in “utility” instead of “constructor” in “parent”
* Label || default
* Arrays
  + Array.sort()
  + Array.push()
  + Array.indexof, -1
* Chaining
  + s
* Using .call() and .apply when making the call to the call back function??
* string.split --> string to an array
* JSON.parse --> well formed JSON string to JS Object
* JSON.stringify. --> JS Object (like array) to JSON string. object,replacer,space, - replace is call for each values in object,
* Asynchronous: Callback
  + When you pass the callback function as a parameter to another functions(containing), the parameters of the call back function are not yet defined.
  + These have to be global or defined in the function(containing) to which the callback function is passed. The parameters have to be passed when the callback function is invoked in the containing application
  + You can pass multiple call back functions - successcallback, errorcallback, completeclallback, next.
    - if (typeof callback === "function")
  + When you use callback
    - Telling Javascript or calling function to execute rest of the code at a latter time
    - You are putting a lot of trust on the functions being called as the function can call back several times and send parameters as they wish. Inversion of control and implicit control issues
    - cb(err, data)
* Events
  + Non-blocking code can be implemented using Events also.
    - Function can emit an event (e.g. “ready”) when it’s the blocking I/O is completed
    - Calling Block/Functions can listen to the event using .on(“ready”, function(){}). On receiving the event the call back function will be called.
  + alternate syntax ()=> {}
* “Tracer” for logging
* If module is imported multiple time within the same project, then the instance of the module is shared
* Dependency Injection for encapsulated code
* “=>” before brace instead of “function” keeps the “this” keyword in the call back. “Self” will not be needed
  + Singleton Pattern
  + Observer Pattern
  + Model Factory
  + Reflection
  + Interfaces
  + Delegates

## Generators

* + Allow writing synchronous code in asynchronous functions. Hence no CallBack block or code is required
  + “yield” keyword is used to make the function call. Program pauses till generator is called again. Generator can be called from within the function call to resume flow
  + Attempt to express asynchronous code in synchronous fashion
  + Pause it self in the middle of program and iterator can resume it
  + function\* gen() { step1 ; yield null; step 2} //yield use to pause and pass message(null) back to iterator
  + var it = gen() //returns an iterator
  + it.next() ; //step 1 and pause at “yield”
  + gen(10) passes 10 back to the yield point to resume
  + using yield you can pass in and pass out values

## Promises

* + Calling a function and subscribing to the ‘completion’ event of the function
  + return new Promise(cb(resolve, reject) { resolve(d}) //d value to be returned along with promise
  + .then helps to chain promises
    - Waits for the promise to resolve and then calls the next method passed as a parameter to the ‘then’
  + “Resolve” – passes data to the next promise in the chain
  + .gate() executes things in parallel but waits for all of them to finish
  + CSP generators
  + No inversion of control as instead of passing callback/control to function/api we hold on to the promise and get a notification when the function is completed
  + getFile(file) {

return new Promise(

function (resolve) {

fakeAjax(file,resolve)

}

} );

}

* + Var p1 = getFile(“File1”)
  + P1.then(output).then(function () {return p2 }
  + .[“file1”,”file2”,”file3”]

.map(getFile)

.reduce() //starts with an initial state and

* Modernizer is JS library that detects HTML5 and CSS3 features in a user’s browser
* Code should say ‘what’ and Comment should say ‘why’ and/or the ‘how’
* Chrome
  + Console
  + Snippets
  + Workspace
  + Inspect Element
* Statements,Expressions, Snippets,
* “falsy values” : 0, -0, false, null, undefined, Nan, “”
* Strings can be in single quotes or double quotes
* Operators
  + === (thriple equals is “strict” equal)
* Function