**Advance JavaScript**

**Call back hell**

The problem in JavaScript is that the only way to "freeze" a computation and have the "rest of it" execute latter (asynchronously) is to put "the rest of it" inside a callback.

For example, say I want to run code that looks like this:

x = getData();

y = getMoreData(x);

z = getMoreData(y);

...

What happens if now I want to make the getData functions asynchronous, meaning that I get a chance to run some other code while I am waiting for them to return their values? In Javascript, the only way would be to recode everything that touches an async computation using [continuation passing style](https://duckduckgo.com/l/?kh=-1&uddg=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FContinuation-passing_style):

getData(function(x){

getMoreData(x, function(y){

getMoreData(y, function(z){

...

});

});

});

I don't think I need to convince anyone that this version is uglier than the previous one. :-)

## What are callbacks?

Callbacks are just the name of a convention for using JavaScript functions. There isn't a special thing called a 'callback' in the JavaScript language, it's just a convention. Instead of immediately returning some result like most functions, functions that use callbacks take some time to produce a result. The word 'asynchronous', aka 'async' just means 'takes some time' or 'happens in the future, not right now'. Usually callbacks are only used when doing I/O, e.g. d

ownloading things, reading files, talking to databases, etc.

When you call a normal function you can use its return value:

**var** result **=** multiplyTwoNumbers(5, 10)

console.log(result)

// 50 gets printed out

However, functions that are async and use callbacks don't return anything right away.

**var** photo **=** downloadPhoto('http://coolcats.com/cat.gif')

// photo is 'undefined'!

In this case the gif might take a very long time to download, and you don't want your program to pause (aka 'block') while waiting for the download to finish.

Instead, you store the code that should run after the download is complete in a function. This is the callback! You give it to the downloadPhoto function and it will run your callback (e.g. 'call you back later') when the download is complete, and pass in the photo (or an error if something went wrong).

downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

**function** **handlePhoto** (error, photo) {

if (error) console.error('Download error!', error)

else console.log('Download finished', photo)

}

console.log('Download started')

The biggest hurdle people have when trying to understand callbacks is understanding the order that things execute as a program runs. In this example three major things happen. First the handlePhoto function is declared, then the downloadPhoto function is invoked and passed the handlePhoto as its callback, and finally 'Download started' is printed out.

Note that the handlePhoto is not invoked yet, it is just created and passed as a callback into downloadPhoto. But it won't run until downloadPhoto finishes doing its task, which could take a long time depending on how fast the Internet connection is.

This example is meant to illustrate two important concepts:

* The handlePhoto callback is just a way to store some things to do at a later time
* The order in which things happen does not read top-to-bottom, it jumps around based on when things complete

## How do I fix callback hell?

Callback hell is caused by poor coding practices. Luckily writing better code isn't that hard!

**CALL**

**Description**

A different this object can be assigned when calling an existing function. this refers to the current object, the calling object. With call, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

**Examples**

Using call to chain constructors for an object

You can use call to chain constructors for an object, similar to Java. In the following example, the constructor for the Product object is defined with two parameters, name and price. Two other functions Food and Toyinvoke Product passing this and name and price. Product initializes the properties name and price, both specialized functions define the category.

function Product(name, price) {

this.name = name;

this.price = price;

if (price < 0) {

throw RangeError('Cannot create product ' +

this.name + ' with a negative price');

}

}

function Food(name, price) {

Product.call(this, name, price);

this.category = 'food';

}

function Toy(name, price) {

Product.call(this, name, price);

this.category = 'toy';

}

var cheese = new Food('feta', 5);

var fun = new Toy('robot', 40);

Using call to invoke an anonymous function

In this purely constructed example, we create an anonymous function and use call to invoke it on every object in an array. The main purpose of the anonymous function here is to add a print function to every object, which is able to print the right index of the object in the array. Passing the object as this value was not strictly necessary, but is done for explanatory purpose.

var animals = [

{ species: 'Lion', name: 'King' },

{ species: 'Whale', name: 'Fail' }

];

for (var i = 0; i < animals.length; i++) {

(function(i) {

this.print = function() {

console.log('#' + i + ' ' + this.species

+ ': ' + this.name);

}

this.print();

}).call(animals[i], i);

}

Using call to invoke a function and specifying the context for 'this'

In below example, when we will call greet the value of this will be bind to object i.

function greet() {

var reply = [this.person, 'Is An Awesome', this.role].join(' ');

console.log(reply);

}

var i = {

person: 'Douglas Crockford', role: 'Javascript Developer'

};

greet.call(i); // Douglas Crockford Is An Awesome Javascript Developer

By method of call, we can pass any this in any function, see above example, greet() don’t have any parameter but “I” is getting as this for greet function and this.person and this.role is getting exeucuting.

Understand the apply then note down

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply>

Java script best practices

<https://www.w3.org/wiki/JavaScript_best_practices>

**Best practices**

## Avoid globals

Below code variable will be global in whole file of JS.

var current = null;

function init(){...}

function change(){...}

function verify(){...}

If you do like below [this is kind of Revealing module pattern]

myNameSpace = function(){

var current = null;

function init(){...}

function change(){...}

function verify(){...}

return{

myinit:init,

mychange:change

}

}();

myNameSpace can call inner function by ‘myNameSpace.myinit’ or myNameSpace.myChange()

## Stick to a strict coding style

* 1. Use JSlint for validating the code

## Modularize — one function per task

## Allow for configuration and translation

## Avoid heavy nesting

## Optimize loops

## Keep DOM access to a minimum

## Don’t trust any data

# The Module Pattern

This pattern is used to mimic classes in conventional software engineering and focuses on public and private access to methods & variables. The module pattern strives to improve the reduction of globally scoped variables, thus decreasing the chances of collision with other code throughout an application.

The module pattern is, by far, the most commonly used design pattern and widely accepted in a number of large projects such as [jQuery](http://jquery.com/), [Dojo](http://dojotoolkit.org/), [ExtJS](http://www.sencha.com/products/extjs) and[YUI](http://yuilibrary.com/).

|  |
| --- |
| ( function( window, undefined ) { |
|  |
| // normally variables & functions start with a lowercase letter but with modules, that is not the case. |
| // The general tradition is to start them with a capital letter instead. |
| function MyModule() { |
|  |
| // `this` refers to the instance of `MyModule` when created |
| this.myMethod = function myMethod() { |
| alert( 'my method' ); |
| }; |
|  |
| // note that we still use a function declaration even when using a function expression. |
| // for more information on why, check out: http://kangax.github.io/nfe/ |
| this.myOtherMethod = function myOtherMethod() { |
| alert( 'my other method' ); |
| }; |
|  |
| } |
|  |
| // expose access to the constructor |
| window.MyModule = MyModule; |
|  |
| } )( window ); |
|  |
| // example usage |
| var myModule = new MyModule(); |
| myModule.myMethod(); // alerts "my method" |
| myModule.myOtherMethod(); // alerts "my other method" |

# The Revealing Module Pattern

var MyModule = ( function( window, undefined ) {

function myMethod() {

alert( 'my method' );

}

function myOtherMethod() {

alert( 'my other method' );

}

// explicitly return public methods when this object is instantiated

return {

someMethod : myMethod,

someOtherMethod : myOtherMethod

};

} )( window );

// example usage

MyModule.myMethod(); // undefined

MyModule.myOtherMethod(); // undefined

MyModule.someMethod(); // alerts "my method"

MyModule.someOtherMethod(); // alerts "my other method"

## Advantages

* Cleaner approach for developers
* Supports private data
* Less clutter in the global namespace
* Localization of functions and variables through closures
* The syntax of our scripts are even more consistent
* Explicitly defined public methods and variables which lead to increased readability

## Disadvantages

* Private methods are unaccessible.Some people say that this leads to the inability of unit testing but more often than not, if you're questioning the integrity of a function, it should probably be engineered in such a fashion that exposes the utility of that function publicly, thus making it testable. Given the adaptation of the module pattern by jQuery and their everyday use of QUnit to automate testing, this bullet point isn't really relevant but still listed for documentation purposes.
* Private methods and functions lose extendability since they are unaccessible (see my comment in the previous bullet point).
* It's harder to patch public methods and variables that are referred to by something private.

# 3. The Singleton Pattern

This pattern restricts instantiation of an object to a single reference thus reducing its memory footprint and allowing a "delayed" initialization on an as-needed basis. This isn't too common amongst JavaScript projects today but more common in WordPress (see Eric Mann's in-depth article [here](http://eamann.com/tech/the-case-for-singletons/)).

This article is part of a series called [JavaScript Design Patterns](http://carldanley.com/javascript-design-patterns/).

## Advantages

* Reduced memory footprint
* Single point of access
* Delayed initialization that prevents instantiation until required

## Disadvantages

* Once instantiated, they're hardly ever "reset"
* Harder to unit test and sometimes introduces hidden dependencies

## Example

var mySingleton = ( function( window, undefined ) {

var instance = null;

// revealing module pattern that handles initialization of our new module

function initializeNewModule() {

function myMethod() {

alert( 'my method' );

}

function myOtherMethod() {

alert( 'my other method' );

}

return {

someMethod : myMethod,

someOtherMethod : myOtherMethod

};

}

// handles the prevention of additional instantiations

function getInstance() {

if( ! instance ) {

instance = new initializeNewModule();

}

return instance;

}

return {

getInstance : getInstance

};

} )( window );

// example usage

mySingleton.getInstance().someMethod(); // alerts "my method"

mySingleton.getInstance().someOtherMethod(); // alerts "my other method"

## The Observer Pattern

This pattern implements a single object (the subject) that maintains a reference to a collection of objects (known as "observers") and broadcasts notifications when a change to state occurs. When we don't want to observe an object, we simply remove it from the collection of objects being observed

## Advantages

* Requires deeper-level thinking of the relationship between the various components of an application
* Helps us pinpoint dependencies
* Excellent at decoupling objects which often promotes smaller, reusable components

## Disadvantages

* Checking the integrity of your application can become difficult
* Switching a subscriber from one publisher to another can be costly

Example :

|  |
| --- |
| // build the Subject base class |
| var Subject = ( function( window, undefined ) { |
|  |
| function Subject() { |
| this.\_list = []; |
| } |
|  |
| // this method will handle adding observers to the internal list |
| Subject.prototype.observe = function observeObject( obj ) { |
| console.log( 'added new observer' ); |
| this.\_list.push( obj ); |
| }; |
|  |
| Subject.prototype.unobserve = function unobserveObject( obj ) { |
| for( var i = 0, len = this.\_list.length; i < len; i++ ) { |
| if( this.\_list[ i ] === obj ) { |
| this.\_list.splice( i, 1 ); |
| console.log( 'removed existing observer' ); |
| return true; |
| } |
| } |
| return false; |
| }; |
|  |
| Subject.prototype.notify = function notifyObservers() { |
| var args = Array.prototype.slice.call( arguments, 0 ); |
| for( var i = 0, len = this.\_list.length; i < len; i++ ) { |
| this.\_list[ i ].update.apply( null, args ); |
| } |
| }; |
|  |
| return Subject; |
|  |
| } )( window ); |
|  |
| // setup an object that fetchs stocks |
| function StockGrabber() { |
|  |
| var subject = new Subject(); |
|  |
| this.addObserver = function addObserver( newObserver ) { |
| subject.observe( newObserver ); |
| }; |
|  |
| this.removeObserver = function removeObserver( deleteObserver ) { |
| subject.unobserve( deleteObserver ); |
| }; |
|  |
| this.fetchStocks = function fetchStocks() { |
| // fake fetching the stocks |
| var stocks = { |
| aapl : 167.00, |
| goog : 243.67, |
| msft : 99.34 |
| }; |
|  |
| // notify our observers of the stock change |
| subject.notify( stocks ); |
| }; |
|  |
| } |
|  |
| // define a couple of different observers |
| var StockUpdaterComponent = { |
| update : function() { |
| console.log( '"update" called on StockUpdater with: ', arguments ); |
| } |
| }; |
| var StockChartsComponent = { |
| update : function() { |
| console.log( '"update" called on StockCharts with: ', arguments ); |
| } |
| }; |
|  |
| // example usage |
| var stockApp = new StockGrabber(); |
| stockApp.addObserver( StockUpdaterComponent ); |
| stockApp.fetchStocks(); // console logs: "update" called on StockUpdater with... |
| stockApp.addObserver( StockChartsComponent ); |
| stockApp.fetchStocks(); // console logs: "update" called on StockUpdater with... "update" called on StockCarts with... |
| stockApp.removeObserver( StockUpdaterComponent ); |
| stockApp.fetchStocks(); // console logs: "update" called on StockCharts with... |
| stockApp.removeObserver( StockChartsComponent ); |
| stockApp.fetchStocks(); // does nothing; no observers |

# The Mediator Pattern

The American Heritage dictionary defines the word [mediator](http://www.thefreedictionary.com/mediator) as *"one that mediates, especially one that reconciles differences between disputants."* In this manner, the mediator pattern usually implements a single object that becomes a shared resource through all of the different pieces of an application. It's a higher-level version of pub/sub in that it's commonly used to communicate across the different features of an application in contrast to being used within one feature to communicate with all of the individual pieces of that same feature.

## Advantages

* Reduces the communication relationship from "many-to-many" to "many-to-one"
* Helps us pinpoint dependencies
* Excellent at decoupling objects which often promotes smaller, reusable components

## Disadvantages

* Introduces a single point of failure
* When modules communicate back and forth using a mediator pattern, it tends to become cumbersome and usually results in a clear performance hit. It's best when the mediator is only used to coordinate actions across multiple features and not for communication within the individual features themselves; keep the airways clean!

Example

|  |
| --- |
| var Mediator = ( function( window, undefined ) { |
|  |
| function Mediator() { |
| this.\_topics = {}; |
| } |
|  |
| Mediator.prototype.subscribe = function mediatorSubscribe( topic, callback ) { |
| if( ! this.\_topics.hasOwnProperty( topic ) ) { |
| this.\_topics[ topic ] = []; |
| } |
|  |
| this.\_topics[ topic ].push( callback ); |
| return true; |
| }; |
|  |
| Mediator.prototype.unsubscribe = function mediatorUnsubscrive( topic, callback ) { |
| if( ! this.\_topics.hasOwnProperty( topic ) ) { |
| return false; |
| } |
|  |
| for( var i = 0, len = this.\_topics[ topic ].length; i < len; i++ ) { |
| if( this.\_topics[ topic ][ i ] === callback ) { |
| this.\_topics[ topic ].splice( i, 1 ); |
| return true; |
| } |
| } |
|  |
| return false; |
| }; |
|  |
| Mediator.prototype.publish = function mediatorPublish() { |
| var args = Array.prototype.slice.call( arguments ); |
| var topic = args.shift(); |
|  |
| if( ! this.\_topics.hasOwnProperty( topic ) ) { |
| return false; |
| } |
|  |
| for( var i = 0, len = this.\_topics[ topic ].length; i < len; i++ ) { |
| this.\_topics[ topic ][ i ].apply( undefined, args ); |
| } |
| return true; |
| }; |
|  |
| return Mediator; |
|  |
| } )( window ); |
|  |
| // example subscriber function |
| var Subscriber = function ExampleSubscriber( myVariable ) { |
| console.log( myVariable ); |
| }; |
|  |
| // example usages |
| var myMediator = new Mediator(); |
| myMediator.subscribe( 'some event', Subscriber ); |
| myMediator.publish( 'some event', 'foo bar' ); // console logs "foo bar" |

# The Prototype Pattern

The prototype pattern focuses on creating an object that can be used as a blueprint for other objects through prototypal inheritance. This pattern is inherently easy to work with in JavaScript because of the native support for prototypal inheritance in JS which means we don't need to spend time or effort imitating this topology.

## Advantages

* New objects created from the "skeleton" of an existing object inherit references to existing functions on the prototype chain, thus boosting performance and keeping memory footprints to a minimum.
* Great for an application where the focus is on object creation

## Disadvantages

* Overkill for a project that uses very few objects and/or does not have an underlying emphasis on the extension of prototype chains

## Example

|  |
| --- |
| // build our blueprint object |
| var MyBluePrint = function MyBluePrintObject() { |
|  |
| this.someFunction = function someFunction() { |
| alert( 'some function' ); |
| }; |
|  |
| this.someOtherFunction = function someOtherFunction() { |
| alert( 'some other function' ); |
| }; |
|  |
| this.showMyName = function showMyName() { |
| alert( this.name ); |
| }; |
|  |
| }; |
| function MyObject() { |
| this.name = 'testing'; |
| } |
| MyObject.prototype = new MyBluePrint(); |
|  |
| // example usage |
| var testObject = new MyObject(); |
| testObject.someFunction(); // alerts "some function" |
| testObject.someOtherFunction(); // alerts "some other function" |
| testObject.showMyName(); // alerts "testing" |

# The Facade Pattern

The purpose of the facade pattern is to conceal the underlying complexity of the code by using an anonymous function as an extra layer. Internal subroutines are never exposed but rather invoked through a [facade](http://www.merriam-webster.com/dictionary/facade) which makes this pattern secure in that it never exposes anything to the developers working with it. The facade pattern is both extremely interesting and very useful for adding an extra layer of security to your already minified code. This pattern is extremely useful when coupled with the [revealing module pattern](http://carldanley.com/js-revealing-module-pattern/).

This article is part of a series called [JavaScript Design Patterns](http://carldanley.com/javascript-design-patterns/).

## Advantages

* Enhances security for your web application
* Works well in combination with other patterns
* Easy to implement
* Makes it easy to patch internals
* Provides a simpler public interface
* Proven useful for other major libraries such as jQuery

## Disadvantages

There aren't any real drawbacks as it provides a unified interface to a set of interfaces in a subsystem. As a result, you aren't forced to make any unwanted compromises, thus a win-win. One possible note worth mentioning is that a developer must decide whether the implicit cost of implementation is really worth the abstraction (though this is generally a small footprint).

## Examples

|  |
| --- |
| // a simple facade that masks the various browser-specific methods |
| function addEvent( element, event, callback ) { |
| if( window.addEventListener ) { |
| element.addEventListener( event, callback, false ); |
| } else if( document.attachEvent ) { |
| element.attachEvent( 'on' + event, callback ); |
| } else { |
| element[ 'on' + event ] = callback; |
| } |
| } |

The above is a basic example of the facade pattern.

|  |
| --- |
| var MyModule = ( function( window, undefined ) { |
|  |
| // revealing module pattern ftw |
| function MyModule() { |
|  |
| function someMethod() { |
| alert( 'some method' ); |
| } |
|  |
| function someOtherMethod() { |
| alert( 'some other method' ); |
| } |
|  |
| // expose publicly available methods |
| return { |
|  |
| // in our normal revealing module pattern, we'd do the following: |
| someMethod : someMethod, |
|  |
| // in the facade pattern, we mask the internals so no one has direct access by doing this: |
| someMethod : function() { |
| someMethod(); |
| } |
|  |
| }; |
|  |
| } |
|  |
| } )( window ); |

The above is a more advanced version of the facade pattern that adds security to internal methods.

# The Factory Pattern

This pattern focuses on object creation but differs from other patterns in the[creation category](http://carldanley.com/javascript-design-patterns/#creational-design-patterns) in that it does not require a constructor function. The factory pattern generally supplies an interface for developers to create new objects through the use of the factory rather than invoking the new operator on an object. Imagine that you needed a car door so you might goto a car factory that produces the product you're interested in and ask it to give you what you need. The factory then supervises the creation of the new car door (or object) and gives it to you. This example paints a good picture for how the factory pattern works; you simply ask it for a type of component, it instantiates the component (given it exists), and returns you what you were looking for.

## Advantages

* Makes complex object creation easy through an interface that can bootstrap this process for you
* Great for generating different objects based on the environment
* Practical for components that require similar instantiation or methods
* Great for decoupling components by bootstrapping the instantiation of a different object to carry out work for particular instances

## Disadvantages

Unit testing can be difficult as a direct result of the object creation process being hidden by the factory methods

## Example

function CarDoor( options ) {

this.color = options.color || 'red';

this.side = options.side || 'right';

this.hasPowerWindows = options.hasPowerWindows || true;

}

function CarSeat( options ) {

this.color = options.color || 'gray';

this.material = options.material || 'leather';

this.isReclinable = options.isReclinable || true;

}

function CarPartFactory() {}

CarPartFactory.prototype.createPart = function createCarPart( options ) {

var parentClass = null;

if( options.partType === 'door' ) {

parentClass = CarDoor;

} else if( options.partType === 'seat' ) {

parentClass = CarSeat;

}

if( parentClass === null ) {

return false;

}

return new parentClass( options );

}

// example usage

var myPartFactory = new CarPartFactory();

var seat = myPartFactory.createPart( {

partType : 'seat',

material : 'leather',

color : 'blue',

isReclinable : false

} );

// outputs: true

console.log( seat instanceof CarSeat );

// outputs a CarSeat object with material "leather", color "blue", isReclinable "false"

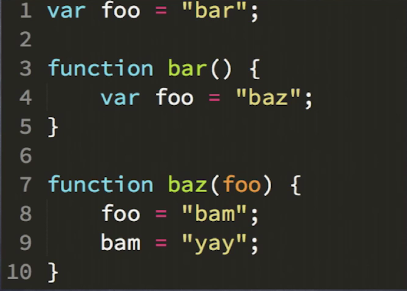
console.log( seat );

**Kyle Simpson JavaScript**

Scope:

* 1. JavaScript is compiler language, but not same way as C++.
  2. It is compile each time whenever it ran.
  3. Its interpreted complier language as same as BASH scripting, if line 3 code is execute, complier don’t know about 4rth line.

Scope - > variable decoration [JavaScript engine working though]

****

**Compilation time: compiler is talking to scope**

var foo = “bar”;

[I got var foo decoration for global scope]

function bar(){

var foo = “baz”;

}

[Hey global scope, I got bar function decoration, add into decoration list]

[Hey scope of bar, I have foo decoration for bar scope]

Function baz(foo){

foo = “bam”;

bam = “yay’;

}

[Hey global scope, I got a baz function decoration, add into decoration list]

[Hey scope of baz, I got a foo decoration for baz scope, add into decoration list]

**Execution phase: [LHS is target and RHS is source]**

**Execution of foo**

foo = “bar”;

[var is removed in decoration time]

[foo is LHS reference, we don’t need to worry about ‘bar’, if we are assigning one variable to another, then only java script engine will ask to scope manager, hey global scope do you have that particular variable reference. But in our example we don’t have]

[Hey global scope (scope manager), I got LHS reference, variable called foo, have you heard of it]

[Answer: Yes, we have him, you declared it couple of micro seconds back]

[by assignment operator , ‘bar’, will be assigned to foo and this line will be executed ]

[Line one is executed now from above screenshot]

**Execution of bar function**

Now **bar** does not exist any more

[Hey scope of bar, I have LHS reference that variable called foo, have you heard him, answer – yes I have, you declared few microseconds back]

[by assignment operator , ‘baz’, will be assigned to foo and this line will be executed ]

**Execution of baz function**

[Hey scope of baz, have heard of foo LHS reference, answer – yes, you have declared just few micro seconds back, and ‘bam’ will be assigned by assignment operator and line is executed now]

**Important point to understand, how JavaScript is creating global variable accidently.**

[hey scope of baz, do you have bam LHS reference, Answer is no I never heard of him, then Java script engine will ask to global scope, hey do you have bam in global scope, Answer is **YES** (even though out side of code don’t have bam variable), basically scope manager have created bam for you, because it don’t have, this is issue when we don’t have variable in LHS side and create accidently, we are in non-strict mode, if we are in strict mode, it will be error, if no declaration of variable]

***Undefined and undeclared are not same, they are not synonyms***

Undefined means var I ; //Uninitialised

Undeclared I = 0; or I;

Another example

Var foo = “bar”;

Function bar(){

Var foo = “baz”; ***//if global variable and inner function variables are same, we them shadowing***

Function baz(foo){

foo = “baz”;

bam = “yay”;

}

Baz();

}

//Below all variable/functions call are RHS, because they are not LHS. LHS comes first so they are RHS.

bar();

foo; //bar

bam; //yay

baz(); //Error – reference error

Difference between function declaration and function expression

Var foo = function bar(){

Var foo = “baz”;

Function baz(foo){

foo = bar;

foo; ///Function

}

Baz();

};

foo(); **/ this will execute because this function expression call.**

bar(); **// Error – this because of we can call function like this because it was function expression not a function declaration.**

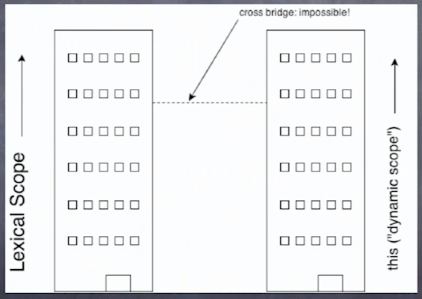
**Anonymous function – three major issues**

* 1. It don’t have self-reference object, if we want to use it in recursion, we cannot do that.
  2. Debugging time stack pointer will not go point in right place because it’s anonymous function, but if you give a name, you can see the error in right place.
  3. Want to check what this function do, we have to check the whole definition of code, but if have function name, we can understand based on name of function.

Note: Suggested by Kyle, go for name function express as mentioned in above example.

**Lexical scope**

Lexical scope – it is compile time scope. Going up by elevator one to another on towards to top. That’s called lexical scope.



Eval : We should not use eval, this is security risk and losing the performance.

Settimeout :

setTimeout("alert('Boom!');", 2000);

This is very unsafe, this forcely use eval which will performance issue.

With : Example

Var obj = {

a:2,

b : 3,

c:4

}

Obj.a = obj.a+obj.b;

Obj.C = obj.a+obj.b;

We can do with WITH keywork so we will not use obj again and again

With (obj){

A = b+c;

C=a+b;

D=3; //Here I am trying create another property for obj, but by execution time, JavaScript engine will create lexical variable in global.

}

So with is more dangerous that eval, because eval is changing lexical scope mechanism in run but with is creating global variable in run time.

Note : In strict mode : with is disallowed to all together.

**LET keyword in JavaScript**

* 1. Var keyword if we use in for loop that will accessible of outside of for loop also.
  2. So we need to use Let keyword and that is not accessible outside of for loop as expected.

ES6 let

let (x = "foo") {

console.log(x); // "foo"

}

console.log(x); // Reference Error!

ES3 or 5

{ let x = "foo";

console.log(x); // "foo"

}

console.log(x); // Reference Error!

If you don't plan to use another ES6 transpiler, you can make *let-er* target ES3+ compatibility, such that the code produced would be:

try{throw "foo"}catch

/\*let\*/ (x /\* = "foo" \*/) {

console.log(x); // "foo"

}

console.log(x); // Reference Error!0

**Hoisting**

|  |  |
| --- | --- |
| After running the code | Complied code |
| a;  b;  var a = b;  var b = 2;  b;  a; | Var a;  Var b;  A=b;  B=2;  B;  A;  **Moving var a,b is called hoisting** |

Note : C++ do need to have manually header file because c++ compiler don’t do automatic hosting, putting manually header file is called hoisting, java script engine do automatic hoisting in compile time.

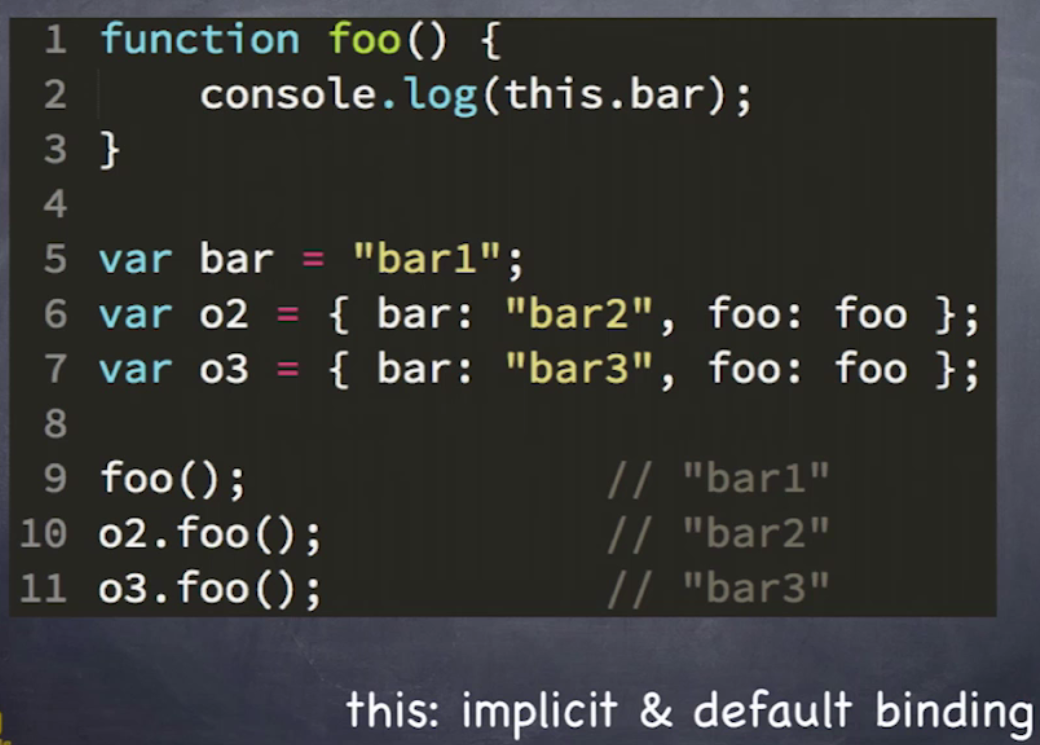
LET DON’T DO HOIST

Variable declaration goes up when hoisting happened but initialization will not move .

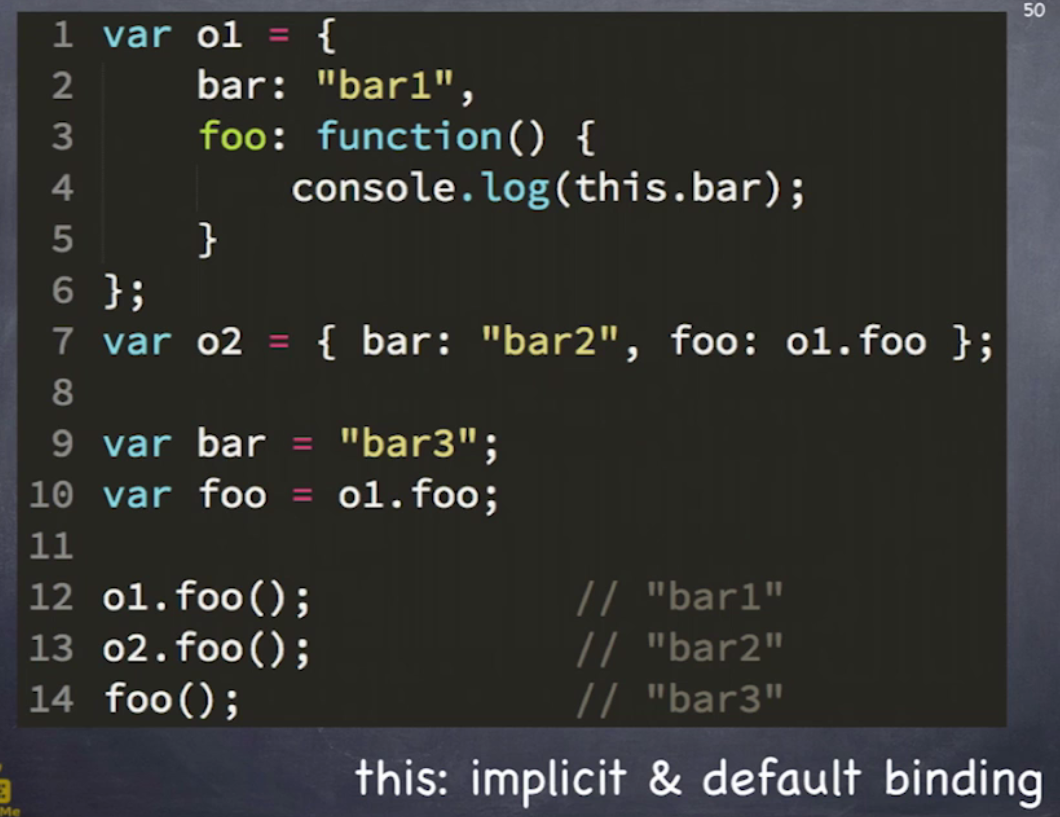
**This KEYWORD**

**Every function, while executing, has a reference to, it’s current execution context, called this. JavaScript’s version of ‘dynamic scope’ is this.**

Ex. If anyone want to go any building, that needs to have address, and **this** keyword can go by reverence or address assignment.



From object we can call function and with THIS keyword this.bar executed with the right values in console.



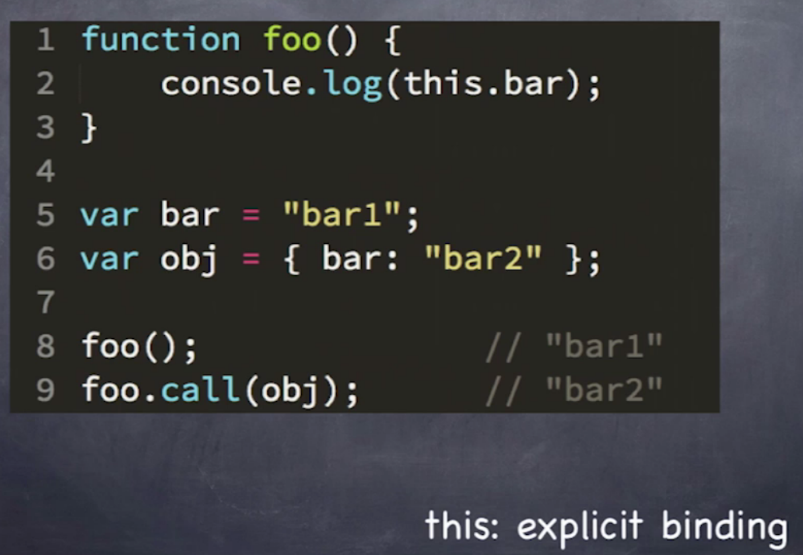
Foo();//bar3 - this is because bar3 was assigned globally.

We cannot manually use lexical scope variable into this, both are different technique, we cannot make bridge, that imposable.

**This – explicit binding**

Below is explicit biding, when we want to call function with different **this object,** we can do by call or apply.

See below: if we obj2 with different value that will be pass into foo function. **Don’t forget this, it’s tricky in interview**



**This – hard binding**

In below code, foo is assign back with new overridden function, which call foo function itself and with obj constant object. If even though we pass obj2 different object in call parentheses, then also bar will print because foo’s overridden code will execute with original obj.

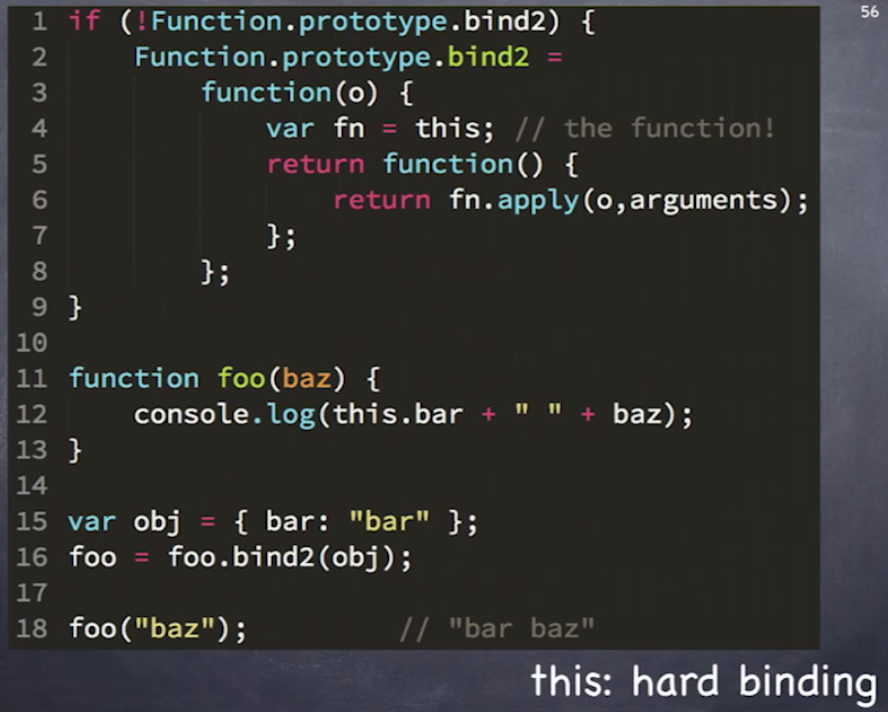


Whenever we need to bind one constant object with function call , we can do above technique to do that.

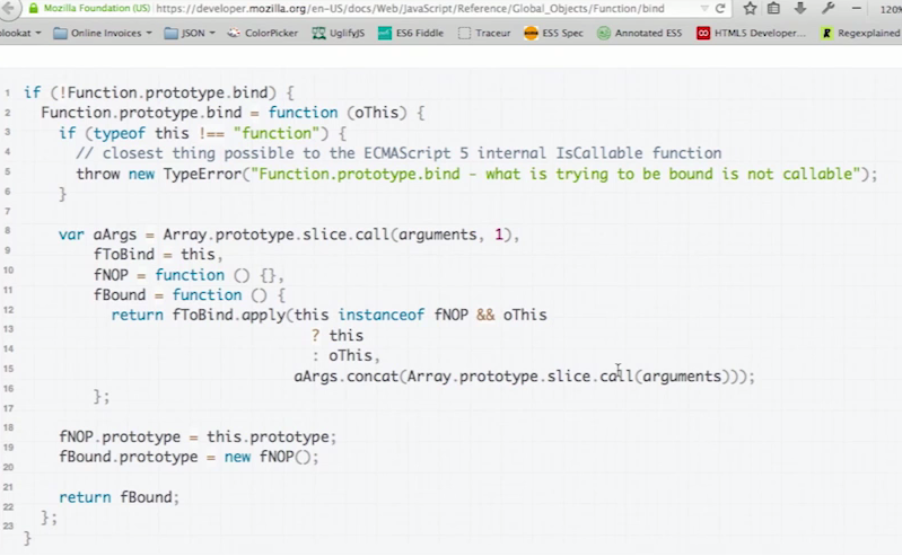
And we can make utility for generic purpose.



Extended version:



Bind Polly fill :



**This – New**

* 1. We can compare new keyword with C++ or java, JavaScript new keyword is having different mechanism here.
  2. You can use new keyword any function which will become constructor call.

Four things will occur when new key word will evoke

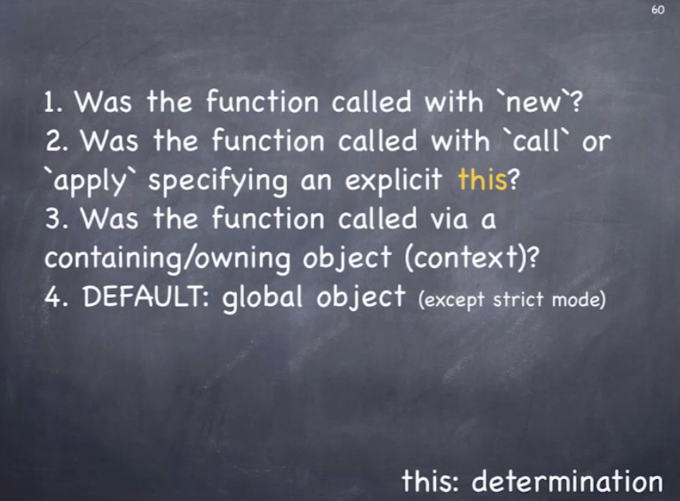
* + 1. Brand new empty object is created
    2. \*\*\* Brand new objected created and linked with some other \*\*

\*\* Means is this in later slides [**A constructor makes objects linked to its own prototype**]

* + 1. Brand new poof object get bound as this keyword that purposes of the function call.
    2. Brand new poof object will implicitly return for us the porpoises of the call.

Note : if you add new keyword (new foo()), it will same as before but it will four above things addition and work line constructor when it will excite by new keyword.

New keyword can override the hard binding.



|  |  |
| --- | --- |
|  | Answers :   * 1. New, call/apply, implicit binding, global   2. NA   3. By call/apply overwide in different fun   4. By bind and have generic overridden method |

## Closure

## 

|  |  |
| --- | --- |
|  | Baz is getting called outside of foo still able to print bar from its lexical scope, that’s called closure. |
|  | Bam is calling foo()(), first foo() is return object of foo function and second () is getting called inner function and inner function is able to print bar from its lexical scope, that’s called closure. |
|  | setTimeout is printing the value bar , that value is coming from lexical scope, that’s closure, if values is change in middle on execution, new value will be printed. Its live connection. |
|  | Any event hander accessing values outside of event handler, that’s due to concept of closure. |
|  | By virtue of the closure foo object can access direct bar property.  **But by definition of closure this is not closure because this don’t have function which remember lexical scope.** |

## Classic module pattern

## Classic module pattern needs to have two properties.

1. Classic module pattern needs to have function expression for IFFY.
2. This pattern needs to have return one or more inner function.

## 

## 

## 

## 

## Work same way as require JS work

## 

## 

## 

## 

## Question 2: Create “native” methods

Define a repeatify function on the String object. The function accepts an integer that specifies how many times the string has to be repeated. The function returns the string repeated the number of times specified. For example:

console.log('hello'.repeatify(3));

Should print hellohellohello.

### Answer

A possible implementation is shown below:

String.prototype.repeatify = String.prototype.repeatify || function(times) { var str = ''; for (var i = 0; i < times; i++) { str += this; } return str; };

## Question 3: Hoisting

What’s the result of executing this code and why.

function test() { console.log(a); console.log(foo()); var a = 1; function foo() { return 2; } } test();

### Answer

The result of this code is undefined and 2.

The reason is that both variables and functions are [hoisted](http://www.sitepoint.com/back-to-basics-javascript-hoisting/) (moved at the top of the function) but variables don’t retain any assigned value. So, at the time the variable a is printed, it exists in the function (it’s declared) but it’s still undefined. Stated in other words, the code above is equivalent to the following:

function test() { var a; function foo() { return 2; } console.log(a); console.log(foo()); a = 1; } test();