Callback function	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	
Call	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.	
Java Script best practices	**Call things by their name — easy, short and readable variable and function names**	isOverEighteen

	Avoid globals	var myNameSpace = {
		current:null,
		init:function(){...},
		change:function(){...},
		verify:function(){...}
		}

	Stick to a strict coding style	'use strict'
	Comment as much as needed but not more	
	Avoid mixing with other technologies	Do not add .CSS in java script functions Ex. color, background ,try to add a class name and pass the class name which is written in .CSS file. Ex. eliement.className = 'error'
	Use shortcut notation when it makes sense	Bad code `var awesomeBands = new Array();`

		awesomeBands[0] = 'Bad Religion'; awesomeBands[1] = 'Dropkick Murphys'; awesomeBands[2] = 'Flogging Molly' Good Code: var awesomeBands = ['Bad Religion', Dropkick Murphys', 'Flogging Molly',];
	Modularized – One function one task, should not have two or multiple things in one function,	It helps to other developers for debug and doing development
	Enhance progressively	Progressive Enhancement as a development practice is discussed in detail in the Graceful degradation versus progressive enhancement. In essence what you should do is write code that works regardless of available technology. In the case of JavaScript, this means that when scripting is not available (say on a BlackBerry, or because of an over-zealous security policy) your web products should still allow users to reach a certain goal, not block them because of the lack of JavaScript which they can't turn on, or don't want to.
	Allow for configuration and translation	config = { CSS:{ /* IDs used in the document. The script will get access to the different elements of the player with these IDs, so if you change them in the HTML below, make sure to also change the name here! */ IDs:{ container:'eytp-maincontainer', canvas:'eytp-playercanvas',

		player:'eytp-player', controls:'eytp-controls',
	Avoid heavy nesting	If you multiple loop in one function, then break it two or multiple functions.
	Optimize loops	var names = ['George','Ringo','Paul','John']; var all = names.length; for(var i=0;i<all;i++){ doSomeThingWith(names[i]); } here do not use names.length in loop, otherwise every time JavaScript needs to read length of array.
	Keep DOM access to a minimum	
	Don't yield to browser whims	Writing code specific to a certain browser is a sure-fire way to keep your code hard to maintain and make it get dated really quickly. If you look around the web you'll find a lot of scripts that expect a certain browser and stop working as soon as a new version or another browser comes around. This is wasted time and effort — we should build code based on agreed standards as outlined in this course of articles, not for one browser. The web is for everybody, not an elite group of users with a state-of-the-art configuration. As the browser market moves quickly you will have to go back to your code and keep fixing it. This is neither effective nor fun.
	Don't trust any data	Check of object and null or blank

Name of Module patterns	Definition	Code example

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	(function(window, undefined) {
		function myModule{
		this.myMethod = function(){
		console.log('Hello my func');
		}
		}
		window.MyModule = myModule;

	strives to improve the reduction of globally scoped variables, thus decreasing the chances of collision with other code throughout an application.	`})(window)`

The Revealing Module Pattern	Adwantages
	• Cleaner approach for developers
	• Supports private data
	• Less clutter in the global namespace
	• Localization of functions and variables through closures
		`var MyModule = ( function( window, undefined ) {`
		`  function myMethod() {`
		`  alert( 'my method' );`
		`  }`
		`  function myOtherMethod() {`
		`  alert( 'my other method' );`
		`  }`
		`  return {`
		`  someMethod : myMethod,`
		`  someOtherMethod : myOtherMethod`
		`  };`
		`} )( window );`
		
		`MyModule.someMethod(); // alerts "my method"`
		`MyModule.someOtherMethod(); // alerts "my other method"`

The Singleton Pattern	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
	•
		function getInstance() {
		if(! instance) {
		instance = new initializeNewModule();
		}
		return instance;
		}
		
		return {
		getInstance : getInstance
		};

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	Subject.prototype.observe = function observeObject(obj) {
		
		this._list.push(obj);
		
		Subject.prototype.unobserve = function unobserveObject(obj) {this._list.splice(i, 1);
		
		Subject.prototype.**notify** = function notifyObservers() {

		var args = Array.**prototype**.**slice**.call(arguments, 0); **this._list[i].update.apply(null, args**

The Mediator Pattern	Observer : Enables notification of a event in one object to different set of objects (instances of different classes)
	
	Mediator: Centralize the communication between set of objects, created from a particular class.
		`Mediator.prototype.subscribe`
		
		`Push use and add into collection`
		
		`Mediator.prototype.unsubscribe`
		
		`Mediator.prototype.publish`
		
		`Use apply function with new object`

Prototype pattern	The prototype pattern focuses on creating an object that can be used as a blueprint for other objects through prototypal inheritance.	
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 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.	

Factory		`function CarDoor`
		`function CarSeat`
		
		`function CarPartFactory() {}`
		`CarPartFactory.prototype.createPart = function createCarPart( options ) {`
		`  var parentClass = null;`

<table>
<tr>
<td></td>
<td></td>
<td>

```
  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

} );
```

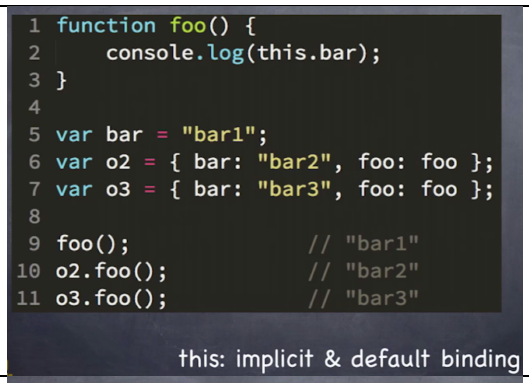
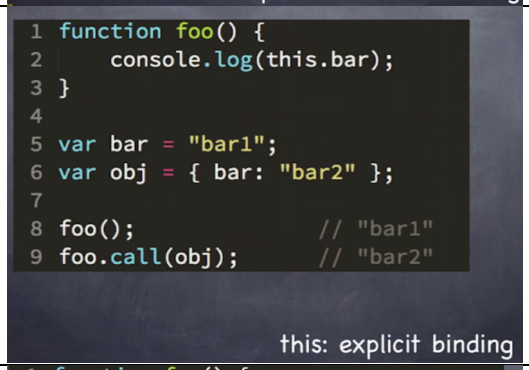
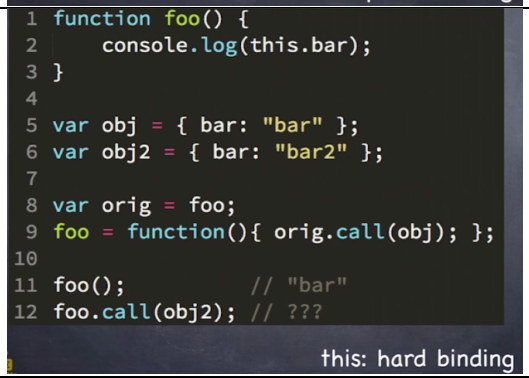
</td>
</tr>
<tr>
<td>Hoisting
Note : C++ do need to have manually header file because c++ compiler don’t do automatic hosting, putting manually header file is called hoisting, java script engine do automatic hoisting in compile time.</td>
<td>

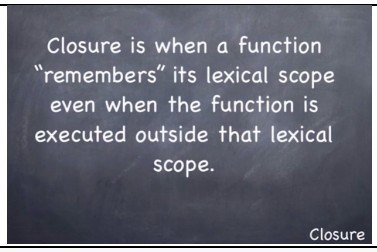
```
After running the code
a;
b;
var a = b;
var b = 2;
b;
a;
```

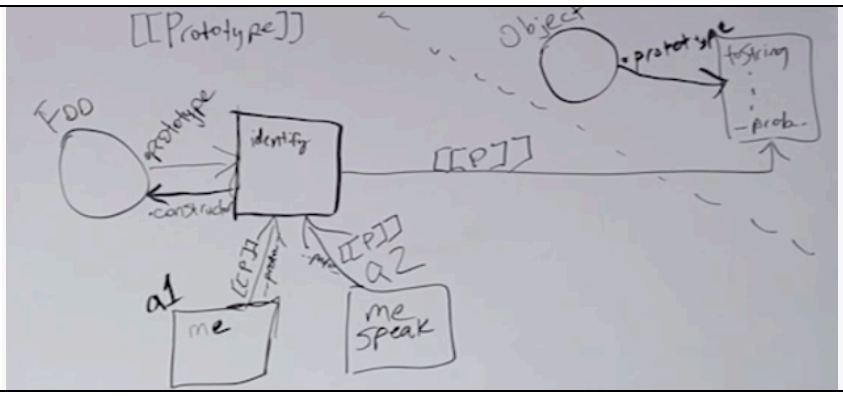
</td>
<td>

```
Complied code
Var a;
Var b;
A=b;
B=2;
B;
A;
```

Moving var a,b is called hoisting
</td>
</tr>
<tr>
<td>This</td>
<td colspan="2">Every function, while executing, has a reference to, it’s current execution context, called this. JavaScript’s version of ‘dynamic scope’ is this.</td>
</tr>
</table>

Implicit and default binding	
Explicit binding	
Hard binding	
NEW keyword	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. [**A constructor makes objects linked to its own prototype**] 3. Brand new poof object get bound as this keyword that purposes of the function call. 4. Brand new poof object will implicitly return for us the purposes 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.
Closer	
datatypes are supported in Javascript	• Undefined • Number • String • Boolean • Object • Function • Null
OOO	 [[prototype]] : its internal linkage __proto__ : this is publically linkage to prototype chain and called dunder proto. It is not slandered until ES6, I-11 supports it but older version doesn't, all other browser adapted.

		 [[Prototype]] Object prototype toString -prob.. FOO prototype identify constructor [[P]] [[P]] [[P]] a2 a1 me me speak