* If any variable is initialized without declaration at LHS, it will be declared in global scope.
* **eval** evaluates the string as expression but it restricts JS compiler to do the code optimization.
* You can pass object in **with** keyword for reference:

var obj = { a:2, b:3, c:4 }

obj.a = obj.b + obj.c;

obj.c = obj.b - obj.a;

or,

with(obj){

a=b+c;

c=b-a;

}

* Even if variables are declared after its usage, they are hoisted to top of the code by compiler.
* **this** and **lexical** scope are not same and cannot be overlapped.
* **this** keyword context can be defined by following 4 rules:

1. Use **new** keyword: It will replace this reference with function itself.
2. Explicit binding: if you use **call** or **apply** method, **this** will always refer to object which is passed as first parameter.
3. Implicit binding rule: object at the call side takes precedence.
4. Default context of method where it is used or global.

* Javascript provides **bind** method to hard bind given object to function.
* **Closure** is when a function “remembers” its lexical scope even when it is executed outside that lexical scope. So, when a function is passed to another function, it still remembers its context.
* **let** keyword in for loop allows variable to link with their call values and it creates a new reference every time.
* **Module** pattern must have outer wrapping method like IIFE and there must be one or more functions should be returned.
* Through **export** and **import**/**module** keywords, we can expose any js file and import it into another file.
* To create object linkage, use following code:

function **Bar**(who){

Foo.call(**this**, who);

}

Bar.prototype = **Object**.**create**(Foo.prototype);

* To delegate the responsibility from one object to other:

var Foo = {

init : function(who){

this.me = who;

},

identify : function(){

return "I am " + this.me;

}

}

var Bar = **Object.create**(Foo);

Bar.speak = function() {

console.log("Hello, " + this.identify() + ".");

}

var b1= Object.create(Bar);

b1.init("b1");

b1.speak();

* Generator function ca be defined as:

function\* gen()

* When you call generator function, it create iterator object which can be triggered next() method. Next() will execute generator method till it faces **yield null;** in the generator method. It will stop till next() function is call again.
* Following is the example of generator and yield:

var run = function\*(){

var x = 1 + (yield null);

var y = 1+ (yield null);

yield (x + y);

};

run();

run(10);

console.log("Meaning of life: " + run(30).value);

yield null means, it will return null to its caller method and passing value to generator method will replace yield.

* Promise in javascript reverses inversion of control i.e. control comes back to main program for execution. Promise is like McD order where you get receipt first and then food in return of that receipt. When you call resolve, promise call the then function.

function getData(d){

return new Promise(function(resolve, reject){ setTimeout(function(){resolve(d);}, 1000);

});}

var x;

getData(10).then(function(num1){ x = 1 + num1; return getData(30);})

.then(function(num2){ var y = 1 + num2; return getData("Meaning of life: " + (x+y));})

.then(function(answer){ console.log(answer);});