**Difference between var and let keyword**

|  |  |
| --- | --- |
| **var** | **let** |
| “var” is there in js from the beginning | “let” was introduced in ES2015 / ES6 |
| “var” has functional scope | “let” variable has block scope |
| “var” gets hoisted  Variable declared with var gets hoisted on top of its functions | “let” not gets hoisted |

**Difference between “==” and “===”**

|  |  |
| --- | --- |
| == | === |
| Comparison operator | Comparison operator |
| Compare values only | Compare value and type |
|  |  |
|  |  |

**Difference between let and const keyword**

|  |  |
| --- | --- |
| let | const |
| With let variable we can change value | Define Constant value i.e. can be change  With array /object const allow to modify value but not allow to reassign value |
|  |  |
|  |  |
|  |  |

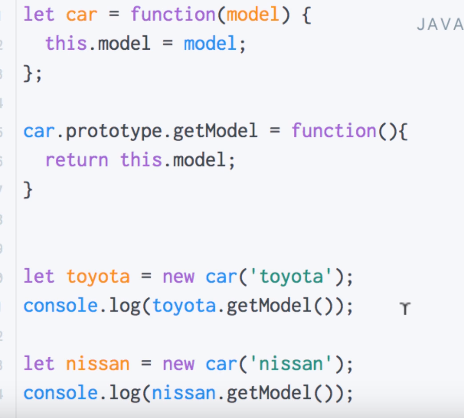
**Difference between undefined and null keyword**

|  |  |
| --- | --- |
| undefined | null |
| Represent empty value | Represent empty value |
| typeof(undefined) => undefined | typeof(null) => object |
|  |  |
|  |  |

**Use of Arrow Functions?**

**What is prototypal inheritance?**

Every object has property called prototype. We can add methods and property to it and when you create other object from this object, the newly created object automatically inherited the property of parents.



**Difference between function declaration and function expression?**

**function expression - Anonymous function saved into the variable.**

**//**Execute function before its declaration

Console.log (funcE ()); //error

let funcE = function () {

console.log(‘function expression’);

}

**function declaration**

//Execute function before its declaration

Console.log (funcD ());

// available before declaration

function funcD () {

console.log(‘function declartion’);

}

**What is promises and why do we use it?**

**setTimeout () ?**

setTimeOut (function () {

console.log(‘a’);

},0);

Console.log(‘b’);

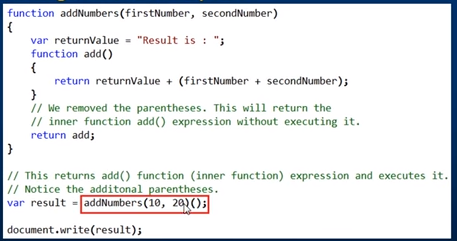
Console.log (‘c’);

**What is a closure and how do you use it?**

Closures are frequently used in JavaScript for object data privacy, in event handlers and callback functions, and in partial applications, currying, and other functional programming patterns.

Objects are not the only way to produce data privacy. Closures can also be used to create stateful functions whose return values may be influenced by their internal state.

A closure is an inner function that has access to the outer functions variable in addition to its own variable and global variable. In simple terms a closure is function inside function. These function, that is the inner and outer functions could be named functions or anonymous functions.



**4 JavaScript Design Patterns You Should Know**

The design patterns:

* Module design patterns
* Prototype design patterns
* Observer design patterns
* Singleton design patterns

**Module design patterns**

JavaScript modules are the most prevalently used design patterns for keeping particular pieces of code independent of other components. This provides loose coupling to support well-structured code.

Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes - that is, a closure that protect variables and methods (however, it will return an object instead of a function). This is what it looks like:

(function () {

// declare private variables and/or functions

return {

// declare public variables and/or functions

}

} )();

var HTMLChanger = (function () {

var contents = 'contents'

var changeHTML = function() {

var element = document.getElementById('attribute-to-change');

element.innerHTML = contents;

}

return {

callChangeHTML: function () {

changeHTML();

console.log(contents);

}

};

})();

HTMLChanger.callChangeHTML(); // Outputs: 'contents'

console.log(HTMLChanger.contents); // undefined

Notice that callChangeHTML binds to the returned object and can be referenced within the HTMLChanger namespace. However, when outside the module, contents are unable to be referenced.

**Revealing Module Pattern**

A variation of the module pattern is called the Revealing Module Pattern. The purpose is to maintain encapsulation and reveal certain variables and methods returned in an object literal. The direct implementation looks like this:

var Exposer = (function() {

var privateVariable = 10;

var privateMethod = function() {

console.log('Inside a private method!');

privateVariable++;

}

var methodToExpose = function() {

console.log('This is a method I want to expose!');

}

var otherMethodIWantToExpose = function() {

privateMethod();

}

return {

first: methodToExpose,

second: otherMethodIWantToExpose

};

})();

Exposer.first(); // Output: This is a method I want to expose!

Exposer.second(); // Output: Inside a private method!

Exposer.methodToExpose; // undefined

**Prototype design patterns**

Any JavaScript developer has either seen the keyword prototype, confused by the prototypical inheritance, or implemented prototypes in their code. The Prototype design pattern relies on the JavaScript prototypical inheritance. The prototype model is used mainly for creating objects in performance-intensive situations.

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype = {

go: function() {

// Rotate wheels

},

stop: function() {

// Apply brake pads

}

}

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype.go = function() {

// Rotate wheels

}

TeslaModelS.prototype.stop = function() {

// Apply brake pads

}

**Revealing Prototype Pattern**

Like Module pattern, the Prototype pattern also has a revealing variation. The Revealing Prototype Pattern provides encapsulation with public and private members since it returns an object literal.

var TeslaModelS = function() {

this.numWheels = 4;

this.manufacturer = 'Tesla';

this.make = 'Model S';

}

TeslaModelS.prototype = function() {

var go = function() {

// Rotate wheels

};

var stop = function() {

// Apply brake pads

};

return {

pressBrakePedal: stop,

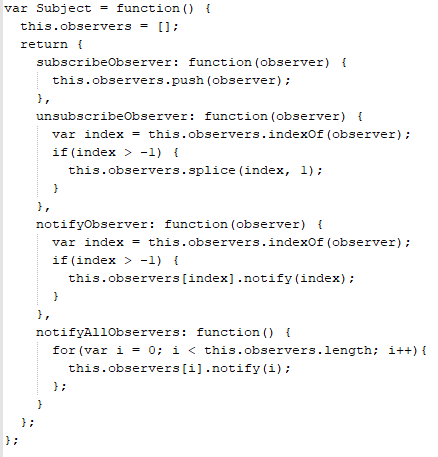
pressGasPedal: go

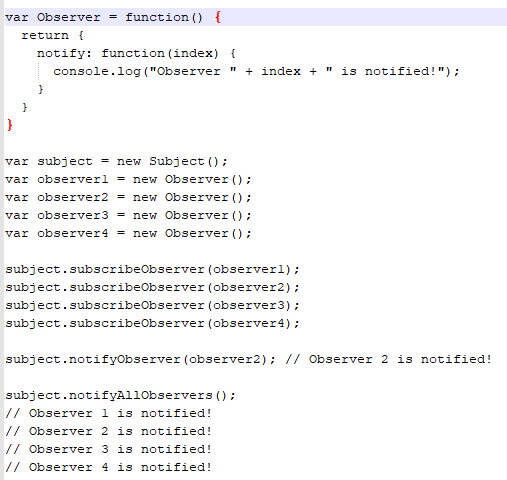
}

}();

**Observer design patterns**

We can create our own Subjects and Observers in JavaScript. Let's see how this is implemented:





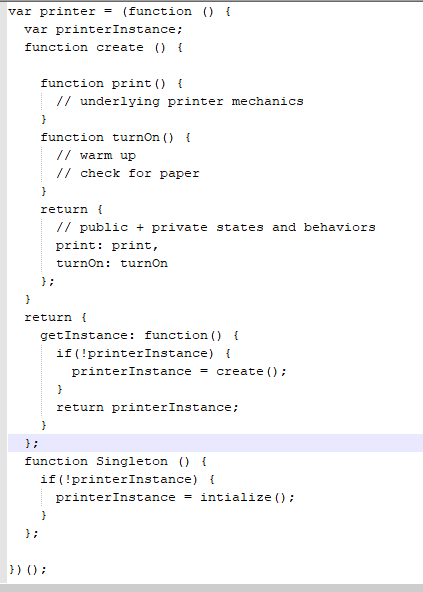
**Publish/Subscribe**

The Publish/Subscribe pattern, however, uses a topic/event channel that sits between the objects wishing to receive notifications (subscribers) and the object firing the event (the publisher). This event system allows code to define application-specific events that can pass custom arguments containing values needed by the subscriber. The idea here is to avoid dependencies between the subscriber and publisher.

Many developers choose to aggregate the publish/subscribe design pattern with the observer though there is a distinction. Subscribers in the publish/subscribe pattern are notified through some messaging medium, but observers are notified by implementing a handler like the subject.

**Singleton design patterns**

A Singleton only allows for a single instantiation, but many instances of the same object. The Singleton restricts clients from creating multiple objects, after the first object created, it will return instances of itself.



var officePrinter = printer.getInstance();

The create method is private because we do not want the client to access this, however, notice that the getInstance method is public. Each officer worker can generate a printer instance by interacting with the getInstance method, like so:

**JavaScript Scoping and Hoisting**

 JavaScript has **function-level scope**.

Function declarations and variable declarations are always moved (“hoisted”) invisibly to the top of their containing scope by the JavaScript interpreter. Function parameters and language-defined names are, obviously, already there. This means that code like this:

function foo () {

var x;

bar ();

x = 1;

}

function foo () {

bar ();

var x = 1;

}

is interpreted like this:

function foo() {

if (false) {

var x = 1;

}

return;

var y = 1;

}

function foo() {

var x, y;

if (false) {

x = 1;

}

return;

y = 1;

}

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted. This is not the case with function declarations, where the entire function body will be hoisted as well. But remember that there are two normal ways to declare functions. Consider the following JavaScript:

function test() {

foo(); // TypeError "foo is not a function"

bar(); // "this will run!"

var foo = function () { // function expression assigned to local variable 'foo'

alert("this won't run!");

}

function bar() { // function declaration, given the name 'bar'

alert("this will run!");

}

}

test();

bind

If you want to use an arbitrary object as value of this, how will you do that?

bind allows you to borrow a method and set the value of this without calling the function. It simply returns an exact copy of the function with new value of this. You can reuse the same function with new value of this without harming the old one.

var ross = {name:'Ross Geller', total:250};

var rossFeeDeductor = monica.deductMonthlyFee.bind(ross, 25);

rossFeeDeductor(); //"Ross Geller remaining balance is 225"

rossFeeDeductor(); //"Ross Geller remaining balance is 200"

rachelFeeDeductor(); //"Rachel Green remaining balance is 900"

var rachel = {name: 'Rachel Green', total: 1500};

var rachelFeeDeductor = monica.deductMonthlyFee.bind(rachel, 200);

rachelFeeDeductor(); //"Rachel Green remaining balance is 1300"

rachelFeeDeductor(); //"Rachel Green remaining balance is 1100"

var monica = {

name: 'Monica Geller',

total: 400,

deductMontlyFee: function(fee){

this.total = this.total - fee;

return this.name + ' remaining balance is '+ this.total;

}

}

 If an older browser dont have bind function, how will you shim it

Function.prototype.bind = Function.prototype.bind || function(context){

var self = this;

return function(){

return self.apply(context, arguments);

};

}

## arguments and call

## Write a simple function to tell whether 2 is passed as parameter or not?

## arguments is a local variable, available inside all functions that provides a collection of all the arguments passed to the function. arguments is not an array rather an array like object. It has length but doesn't have the methods like forEach, indexOf, etc.

## function isTwoPassed(){

## var args = Array.prototype.slice.call(arguments);

## return args.indexOf(2) != -1;

## }

## isTwoPassed(1,4) //false

## isTowPassed(5,3,1,2) //true

## apply - How could you use Math.max to find the max value in an array?

Use apply on Math.max and pass the array as apply takes an array of arguments. Since we are not reading anything from this or using it at all. We can simply pass null as first parameter.

function getMax(arr){

return Math.max.apply(null, arr);

}

console.log(getMax([5,3,1,2,100,300,12.2,100,066,855,955,5,4,3]));

**Note:**

call and apply, both takes the value of this as first parameter. However, call takes a collection of arguments after first parameter whereas apply use an array of arguments as second parameter.

## this

At the time of execution of every function, JavaScript engine sets a property to the function called this which refer to the current execution context. this is always refer to an object and depends on how function is called. There are 7 different cases where the value of this varies.

* In the global context or inside a function this refers to the window object.
* Inside IIFE (immediate invoking function) if you use "use strict", value of this is undefined. To pass access window inside IIFE with "use strict", you have to pass this.
* While executing a function in the context of an object, the object becomes the value of this
* Inside a setTimeout function, the value of this is the window object.
* If you use a constructor (by using new keyword) to create an object, the value of this will refer to the newly created object.
* You can set the value of this to any arbitrary object by passing the object as the first parameter of bind, call or apply
* For dom event handler, value of this would be the element that fired the event

**log prefix - How could you set a prefix before everything you log? for example, if you log('my message') it will log: "(app) my message"**

Just get the arguments, convert it to an array and unshift whatever prefix you want to set. Finally, use apply to pass all the arguments to console.

function log(){

var args = Array.prototype.slice.call(arguments);

args.unshift('(app)');

console.log.apply(console, args);

}

log('my message'); //(app) my message

log('my message', 'your message'); //(app) my message your message

## Scope and hoisting

function foo(){

function bar() {

return 3;

}

return bar();

function bar() {

return 8;

}

}

foo();//?

Output : 8

var a = 1;

function b() {

a = 10;

return;

function a() {}

}

b();

console.log(a);

Output: 1

**Closures Inside Loops**

 This will not output the numbers 0 through 9, but will simply print the number 10 ten times.

for(var i = 0; i < 10; i++) {

setTimeout(function() {

console.log(i);

}, 10);

}

Output : 1o times 10

The console log is inside the anonymous function of setTimeout and setTimeout is executed when current call stack is over. So, the loop finishes and before setTimeout get the chance to execute. However, anonymous functions keep a reference to i by creating a closure. Since, the loop is already finished, the value i has been set to 10. When setTimeout use the value of i by reference, it gets the value of i as 10. Hence, you see 10 ten times.

**Solution:**You can fix it by avoiding closure. Just create a IIFE (Immediately Invoked Function Expression), it will create its own scope and you can pass i to the function. In that case i will be a local variable (will not refer to i in the closure) and value of the i in every loop will be preserved.

for(var i = 0; i < 10; i++) {

setTimeout(console.log.bind(console, i), 10);

}

for(var i = 0; i < 10; i++) {

setTimeout((function(i) {

console.log(i);

})(i), 10)

}

**delete can delete but**

**Look at the code below, I have a property in a object and I am creating a new object where I am setting it to a new value. If I delete that property what will i get if I try to access that property?**

var myObject = {

price: 20.99,

get\_price : function() {

return this.price;

}

};

var customObject = Object.create(myObject);

customObject.price = 19.99;

delete customObject.price;

console.log(customObject.get\_price());

Answer: You will see 20.99

This is very interesting question. When you create object.create(myObject) you are creating new object where the myObject will be the parent of the newly created object. Hence the price property will be at the parent.

When you are assigning some value to customObject.price, you are creating a new property on the child. This means, when you delete customObject.price it deletes the priceproperty in the customObject (in the child). However, when you call the method getprice, first it looks for this.price in the child since the customObject doesn't have price property, JavaScript executor walks through the prototype chain towards the parent. Since customObject was inherited from myObject and myObject has a price property, the get\_price method returns the price from parent. Hence, you are getting 20.99

## Pass by value or by reference

## Does JavaScript pass parameter by value or by reference?

Primitive type (string, number, etc.) are passed by value and objects are passed by reference. If you change a property of the passed object, the change will be affected. However, you assign a new object to the passed object, the changes will not be reflected.

var num = 10,

name = "Addy Osmani",

obj1 = {

value: "first value"

},

obj2 = {

value: "second value"

},

obj3 = obj2;

console.log("obj 3",obj3)

function change(num, name, obj1, obj2) {

num = num \* 10;

name = "Paul Irish";

obj1 = obj2;

obj2.value = "new value";

}

change(num, name, obj1, obj2);

console.log(num); // 10

console.log(name);// "Addy Osmani"

console.log(obj1.value);//"first value"

console.log(obj2.value);//"new value"

console.log(obj3.value);//"new value"

## Memorization

## How could you implement cache to save calculation time for a recursive Fibonacci function?

var memo = [];

function \_fibonacci(n) {

if(memo[n]){

return memo[n];

}

else if (n < 2){

return 1;

}else{

fibonacci(n-2) + fibonacci(n-1);

}

}

## Cache function execution

**How could you cache execution of any function**

You could have a method where you will pass a function and it will internally maintain a cache object where calculated value will be cached. When you will call the function with same argument, the cached value will be served.

function cacheFn(fn) {

var cache={};

return function(arg){

if (cache[arg]){

return cache[arg];

}

else{

cache[arg] = fn(arg);

return cache[arg];

}

}

}

What if you are passing more than one argument?

First we have to use arguments to get all the parameters passed to the function and then we can generate key for the cache object. Generating key for the cache object could be tricky and one solution could be just get the all the parameters and concatenate those. Look at the code below.

return function(){

var args = arguments;

var key = [].slice.call(args).join('');

if(cache[key]){

return cache[key];

}

else{

cache[key] = fn.apply(thi, args);

return cache[key];

}

}

## Animation

How could you implement moveLeft animation?

Use setInterval that will place the element to the left position by some pixels in every 10ms. Hence, you will see the element moving towards the desired position. When you call setInterval, it returns a timeId. After reaching the desired location, you should clear the time interval so that function will not be called again and again in every 10ms.

function moveLeft(elem, distance) {

var left = 0;

function frame() {

left++;

elem.style.left = left + 'px';

if (left == distance)

clearInterval(timeId)

}

var timeId = setInterval(frame, 10); // draw every 10ms

}

## Currying

 How would you implement currying for any functions?

**What is curring:** Curring is partial invocation of a function. Currying means first few arguments of a function is pre-processed and a function is returned. The returning function can add more arguments to the curried function. It's like if you have given one or two spice to the curry and cooked little bit, still you can add further spice to it. A simple example will look like-

function addBase(base){

return function(num){

return base + num;

}

}

var addTen = addBase(10);

addTen(5); //15

addTen(80); //90

addTen(-5); //5

# [**What is the difference between native objects and host objects?**](https://stackoverflow.com/questions/7614317/what-is-the-difference-between-native-objects-and-host-objects)

**Difference between slice, substr, substring**