JavaScript:

Table of Contents

[1. understanding js scope and scoping rules 3](#_Toc54118140)

[1.1. Lexical Scop: JS follows Lexical Scop 3](#_Toc54118141)

[1.2. Variable shadowing and Variable overriding 3](#_Toc54118142)

[1.3. Overriding Global variable 4](#_Toc54118143)

[2. JavaScript Pattern: How to properly define variables in javaScript. Considering Global vs Local, const, block scope. 5](#_Toc54118144)

[3. Object Oriented Programming in JavaScript Series - Part 1 6](#_Toc54118145)

[3.1. Creating objects with constructor functions. 6](#_Toc54118146)

[3.1.1. JavaScript Function Constructors 7](#_Toc54118147)

[4. Prototype ( Object Oriented Programming in JavaScript Series - Part 2) 10](#_Toc54118148)

[JavaScript Prototype Explained 10](#_Toc54118149)

[Overwriting Object Methods 12](#_Toc54118150)

[5. Object.create Method ( Object Oriented Programming in JavaScript Series - Part 3) 12](#_Toc54118151)

[6. Prototype Chain ( Object Oriented Programming in JavaScript Series - Part 4) 14](#_Toc54118152)

[7. Extending Function Constructors: Object Oriented Programming in JavaScript Series - Part 5 16](#_Toc54118153)

[8. Object.setPrototypeOf() Method : Object Oriented Programming in JavaScript Series - Part 6 17](#_Toc54118154)

[Super keyword 19](#_Toc54118155)

[Avoid setting \_\_proto\_\_ manually 19](#_Toc54118156)

[9. Object.assign() Method : Object Oriented Programming in JavaScript Series - Part 7 20](#_Toc54118157)

[10. Mixins in JavaScript : Object Oriented Programming Series - Part 8 23](#_Toc54118158)

[Mixins In JavaScript 23](#_Toc54118159)

[What is a Mixin? 23](#_Toc54118160)

[10.1. Inheritance vs Composition 23](#_Toc54118161)

[Composing Object using Mixins 23](#_Toc54118162)

[Factory Functions 24](#_Toc54118163)

[Factory Function with closures 24](#_Toc54118164)

[Function Mixins 25](#_Toc54118165)

[Conclusion 26](#_Toc54118166)

[11. Javascript Classes ES2015 tutorial : Object Oriented Programming Series - Part 9 26](#_Toc54118167)

[JavaScript Classes 26](#_Toc54118168)

[Class Keyword 26](#_Toc54118169)

[Feels like Classes, Smells like Classes! 27](#_Toc54118170)

[Syntax 27](#_Toc54118171)

[Simulating Class with Function Constructor 28](#_Toc54118172)

[Adding Methods 28](#_Toc54118173)

[12. Javascript extend class es6 : Object Oriented Programming Series - Part 10 31](#_Toc54118174)

[13. Static Methods in JavaScript Tutorial : Object Oriented Programming Series - Part 11 33](#_Toc54118175)

[13.1. 2 types of method : static and instance method 33](#_Toc54118176)

[14. Object Oriented Programming Series - Part 1 34](#_Toc54118177)

[15. Hoisting in JavaScript and function scope issues with keyword var 38](#_Toc54118178)

[16. Javascript Immediately Invoked Function Expression IIFE - Tutorial 42](#_Toc54118179)

[17. Javascript Closure tutorial ( Closures Explained ) 44](#_Toc54118180)

[18. JavaScript Practical Applications of Call, Apply and Bind functions 50](#_Toc54118181)

[19. Async Await JavaScript ES7 53](#_Toc54118182)

[20. Applications of JavaScript Spread Operators Tutorial | ES6 / ES2015 56](#_Toc54118183)

[21. Javascript Default Parameters ES6 ES2015 tutorial 57](#_Toc54118184)

[22. javaScript Method chaining tutorial ( function chaining) 60](#_Toc54118185)

[23. JavaScript Sets and Weaksets | ES2015 62](#_Toc54118186)

[24. JavaScript Maps & WeakMaps 66](#_Toc54118187)

[25. Javascript pass by Value vs pass by Reference tutorial 71](#_Toc54118188)

[26. stack implementation in JavaScript 73](#_Toc54118189)

[27. Big O Notation | time complexity of algorithms 75](#_Toc54118190)

[28. Debounce in Javascript 75](#_Toc54118191)

[29. Throttling in JavaScript 76](#_Toc54118192)

[30. The Differences Between forEach() and map() that Every Developer Should Know 77](#_Toc54118193)

[2. Ability to chain other methods 77](#_Toc54118194)

[31. The JavaScript Reduce Method Explained 78](#_Toc54118195)

[Final Example - Changing an Object Structure 79](#_Toc54118196)

[32. Array.from Method in JavaScript 81](#_Toc54118197)

[33. JavaScript Array Methods , flat , flatMap , reduceRight, copyWithin 82](#_Toc54118198)

[34. JavaScript ES2020 awesome new features. 85](#_Toc54118199)

[Angular 5 Unit Testing with Jasmine Karma and Protractor 92](#_Toc54118200)

[Angular 5 setup with SASS, LESS preprocessor tutorial 93](#_Toc54118201)

# 

# understanding js scope and scoping rules

## Lexical Scop: JS follows Lexical Scop

Scope of a variable is defined by its location within the source code

Nested functions have access to variables declared in their outer scope

var global ="global"

alert(aa) //"Uncaught ReferenceError: aa is not defined"

function a(){

var aa = "aa"

console.log(global) // "global"

}

a()

## Variable shadowing and Variable overriding

var global ="global"

function a(){

global = "overriding global"

alert("from function a()" + global)

}

a()

//output : from function a() overriding global

## Overriding Global variable

var global ="global"

function a(){

global = "overriding global"

/\* alert("from function a()" + global) \*/

}

a()

alert("from after function a() execution " + global)

//output: from after function a() execution overriding global

If I am trying to redefine the variable in function a()

var global ="global"

function a(){

var global = "redefining global"

alert("from function a()" + global)

}

a()

output : from function a()redefining global

but if I try to call it from outside the scope of a it changes the value to original as soon as comes out of a(), it changes to original value

when I redefine a variable, it created a new variable which shadow the gobal variable defined in the global scope

var global = "global";

function a()

var global = "redefining global";

//alert('from funcion a():' + global);

}

a();

alert('from after funcion a() execution:' + global);

//output: from after funcion a() execution:global

# JavaScript Pattern: How to properly define variables in javaScript. Considering Global vs Local, const, block scope.

Lexical Scop: JS follows Lexical Scope: variable defined outside the function is automatically available within the function

If I am not defining a variable and assigned value to it inside the function that makes it automatic global:

(if I console this variable outside the function, I will not get err

If we don’t want that thing to happen : “use strict” prevents making this kind of mistakes: it enforce you to define a variable before assigning it

(reference err)

Block Scope:

Keyword var doesn’t provide block scope: means

Within the function once u define a variable using var it is available throughout the function even inside any block of code. And if you change its value within the block a block it persisted.

'use strict';

(function(){

var i= 1;

if(true){

var i= 2;

console.log(i) //2

}

console.log(i)//2

})()

If I want I to be 2 only within this if condition

'use strict';

(function(){

Let i= 1;

if(true){

let i= 2;

console.log(i) //2

}

console.log(i)//1

})()

Use strict: let is only available in ECMA6

CONST:

const i =1;

i= 3; // assignment to a constant variable

console.log(i) // assignment to a constant variable

# Object Oriented Programming in JavaScript Series - Part 1

JavaScript is considered as a prototype-based Object Oriented Programming language because it uses prototypal inheritance. It’s quite different than the classical OOP languages like Java. Initial version of JavaScript did not have a concept of a class. Functions used with new keyword turned into a constructor, which played a role of a class. When combined with the Object.create() function, JavaScript made prototypal inheritance possible.

The ES2015 JavaScript version introduced few new functions like Object.assign() and Object.setProtoTypeOf(). These functions introduced an easy way to compose objects by using Object Literals. ES2015 also introduced the class keyword. I guess to make all those Java folks more comfortable! Syntactically, JavaScript classes look like as if classical inheritance has finally arrived in JavaScript. However, under the hood it’s using the same function constructors and prototypal inheritance.

## Creating objects with constructor functions.

<http://techsith.com/javascript-function-constructors/>

constructor is used to set the initial properties of value(this)

//Car constructor function

let Car = function(color){

this.color = color

}

Let redCar = new Car(‘red)

Or

If you were to manually create an object without using the constructor function, the syntax will look like following:

let redCar = {

color: 'red'

};

\_\_proto\_\_, is its creator.

If I don’t use the new keyword(constructor without new kw would get undefined) returns undefined

Reasons:

It looks as a regular function if nothing is returns it will be undefined

### JavaScript Function Constructors

It is very important to understand how JavaScript function constructors work before we move to higher level topics of Object Oriented JavaScript. In this article, I am not going to explain about the JavaScript class constructor. Instead, I’ll explain about ‘function’ constructors.  
The ultimate objective of Object Orient Programming is to create objects with ease. Languages like Java have classes and each class can have a constructor, and its job is to set initial properties of the object. In JavaScript, you can create an object by directly using constructors. JavaScript lets you use a function as a constructor, if called in a specific way. And, that’s what this entire article is all about!

Usually, the job of a function is to run a specific job. Let’s say, I want to add two numbers. will write a function that takes two arguments and return the addition of the two numbers.

Sample of a function for adding two numbers:

let add = function(number1, number2) {

return number1 + number2;

};

let sum = add(1, 2);

Now, let’s look at an example of a function constructor.

let Car = function(color) {

this.color = color;

};

let redCar = new Car('red');

As shown in the above example, a regular function takes arguments. It performs an action within the body of the function and returns the result. A regular function may or may not return anything.  
Function constructor, on the other hand, accepts color as an argument and sets it to its this. The constructor function does not contain a return statement. Instead, it returns a new object with the property that is passed as an argument.

The basic syntax difference for executing a regular function and a constructor function is the usage of the new keyword.

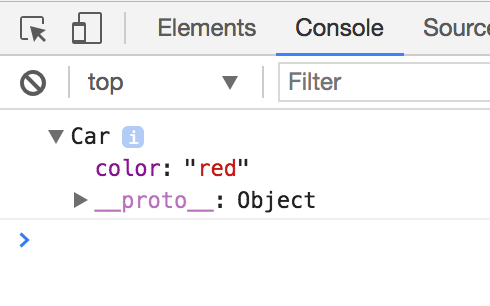
If you were to manually create an object without using the constructor function, the syntax will look like following:

let redCar = {

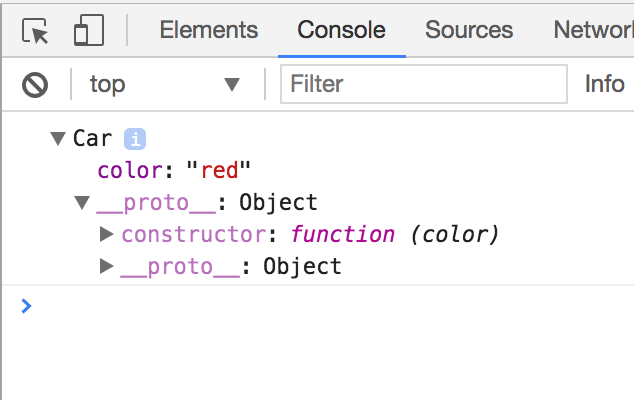
color: 'red'

};

If you inspect the newly created object in a browser, this is how it would look like:

[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-07-at-1.57.08-PM.png)

As you can notice, it has a color property which is set to ‘red’; The property, \_\_proto\_\_, is its creator. If you expand the \_\_proto\_\_, you will see the following:

[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-07-at-1.57.32-PM.png)

The reference to the constructor that set the initial property is the Car constructor that you used to create redCar.

Drawbacks of not using new

Now, let’s consider a scenario where you execute a constructor function as a regular function without using the keyword ‘new’.

let Car = function(color) {

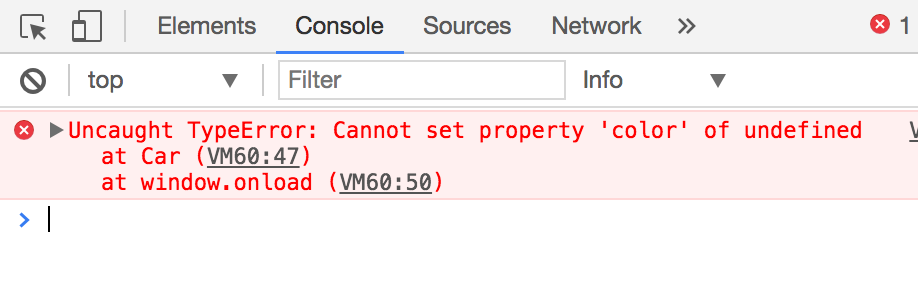
this.color = color;

};

let redCar = Car('red');

In this case, this.color = color will create a new property color in the global space. You can verify the result by using console.log(window.color);

To prevent such error, add se use strict, which throws the following exception:

[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-07-at-5.05.26-PM.png)

You can also use the new.target to throw a customized error message..

let Car = function(color) {

if (!new.target) throw 'Car() must be called with new';

this.color = color;

};

let redCar = new Car('red');

Private Vs Public properties

In JavaScript, all object properties are public, which means you can access it directly. For example, redCar.color. JavaScript does not have a native support for private property. Private properties are the properties you cannot access directly by using a dot notation. However, you can call a getter or have a getter method like getColor to access it and use closure.

let Car = function(\_color) {

this.setColor = function(color){

\_color = color;

}

this.getColor = function() {

return \_color;

};

};

let redCar = new Car('red');

console.log(redCar.getColor());//red

Here \_color is a private variable that cannot be accessed by the object using redCar.\_color.  
Since it is a closure, the methods have access to them.

Adding methods to a Constructor

Best way to have methods is to set methods in prototype space. For more information about settings methods in the prototype space, go to [Prototype](http://techsith.com/javascript-prototype-explained/).

# Prototype ( Object Oriented Programming in JavaScript Series - Part 2)

# JavaScript Prototype Explained

Every constructor function has a property called prototype. You can add properties and methods to it. Using a constructor function, When you create a new object, the object has access to prototype methods.

Sample code:

let Car = function(color) {

this.color = color;

};

Car.prototype.getColor = function() {

return this.color;

};

let redCar = new Car('red');

console.dir(redCar);

If you inspect the redCar object, you will see the following.[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-13-at-9.01.42-PM.png)In the above image, the getColor() is not a part of the redCar object. There is another property called \_\_proto\_\_, which is set automatically. You can think of \_\_proto\_\_ as the creator of redCar. If you expand \_\_proto\_\_, the getColor method and the constructor are displayed. Constructor is the a reference to the Car Constructor that created redCar.  
[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-13-at-9.02.03-PM.png)If you expand the Car constructor, the prototype property of the constructor is displayed. If you further expand the prototype property, the getColor method is displayed.

[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-13-at-9.02.35-PM.png)The getColor method from \_\_proto\_\_ is actually a reference to the one you added to the constructor’s prototype. It looks tricky, however, all you care about is that \_\_proto\_\_, the creator of redCar, has the getColor method. Hence, when you call the redCar.getColor() method, JavaScript will look inside the redCar object. If the method doesn’t exist in the object, it will look for the method in the object’s creator, which is \_\_proto\_\_. It will find it and execute it.  
[](http://techsith.com/wp-content/uploads/2017/07/Screen-Shot-2017-07-13-at-9.03.02-PM.png)So, let’s analyse a little more and discover how is \_\_proto\_\_ created. If you further expand \_\_proto\_\_, you will see its creator, which is the master \_\_proto\_\_. This is basically a master object called Object (the letter ‘O’ is capitalized). Every object in JavaScript is created from Object. It also has some methods. One of the methods is toString(). When you call this method on redCar, JavaScript will first search for this method in the redCar object. If it can’t find it there, it keeps searching for the method all the way to the end of the prototype chain.

There are pros and cons of using prototype inheritance. The best part is, the inheritance makes objects lighter. The drawback is that every time you call a method that is not a part of the object, it looks for the method up the prototype chain. Hence, it makes it a little slower. The workaround is, if you are going to call this method frequently, store it in a variable to avoid the lookup.

let toString = Object.toString;

console.log(redCar.toString());

### Overwriting Object Methods

If you want to customize the toString() method, you can add it to the object’s prototype. However, there are times when you want to make method available to every object. In such case, add it to the master object’s prototype. This would shadow the native toString() method.

Object.prototype.toString = function(){

return `color:${this.color}`;

};

let Car = function(color) {

this.color = color;

};

Car.prototype.getColor = function() {

return color;

};

let redCar = new Car('red');

console.dir(redCar.toString());

In next article, I will show you how to use Object.create() method to set prototype of one object to another. Follow the link below. [Object.create() method](http://techsith.com/object-create-method-javascript/)

# Object.create Method ( Object Oriented Programming in JavaScript Series - Part 3)

Object.create() makes prototype inheritance possible in JS

Every object in jS is created using master object (Object)

**Object**

**(it is a constructor which is a function)**

Syntax

Object.create( protottypeObject, propertyObject)

Ex:

Const myObj = Object.create(Object.prototype)

Output //

{}

--proto--:Object

proto--:it is createor which will point to object prototype

when u create object literal

const myLiteral = {}

this also created using the Object.create METHOD INTERANLLY, a proto type set to objects prototype

--proto--:Object

proto--:it is creator which will point to object prototype

if you want to create object without prototype

const noProto = Object.create(null) // which means don’t set any prototype (null)

output: no properties

const Car = function(color){

this. color = color

}

1. Const car1 = new Car(‘red’);
2. Const car2 = Object.create(Car.prototype);

In 1. I am running the constructor

In 2. I am not running the constructor

Object.create is used to extend the constructor. Compared to classical(it will copy all the method present in base class to sub class) inheritance it will just set ref from subclass to base class

Object .create() which enable to set 1 constructor prototype to other

Diff in 2. Is not setting the initial property constructor is usually set

We can extend one constructor into another constructor

const Car = function(color){

this. color = color

}

Car.prototype ={

getColor(){

return this.color;

}

}

Const toyCar =Function(){

}

toyCar.prototype = Object.create(Car.prototype)

const legoCar = new toyCar()

CONSOLE.LOG(legoCar INSTANCEof toyCar) //true

# Prototype Chain ( Object Oriented Programming in JavaScript Series - Part 4)

// Car constructor

const Car = function() {};

// Set Car's prototype

Car.prototype = {

print() {

return 'this is a Car';

}

};

// ToyCar constructor

const ToyCar = function() {};

// Set ToyCar's prototype to be Car's prototype

ToyCar.prototype = Object.create(Car.prototype);

// Adding ToyCar's own prototype print method

ToyCar.prototype.print = function(){

return 'this is a ToyCar';

}

// Creating LegoCar object from ToyCar constructor

const legoCar = new ToyCar();

console.log(legoCar); //'this is a ToyCar'

// Car constructor

const Car = function() {};

// Set Car's prototype

Car.prototype = {

print() {

return 'this is a Car';

}

};

// ToyCar constructor

const ToyCar = function() {};

// Set ToyCar's prototype to be Car's prototype

ToyCar.prototype = Object.create(Car.prototype);

// Adding ToyCar's own prototype print method

ToyCar.prototype.print = function(){

return 'this is a ToyCar';

}

const ToyTransformer = function() {};

// Set ToyCar's prototype to be Car's prototype

ToyTransformer.prototype = Object.create(ToyCar.prototype);

// Adding ToyCar's own prototype print method

ToyTransformer.prototype.print = function(){

return 'this is a ToyTransformer';

};

# Extending Function Constructors: Object Oriented Programming in JavaScript Series - Part 5

Extending a Function Constructor by Calling Base Constructor from Sub Constructor and Inheriting Properties.

// base constructor

let Maamal = function(legs){

this.legs = legs

}

Maamal.prototype = {

walk(){

return 'walking'

},

sleep(){

return 'sleepng'

}

};

let Bat = function(legs,isVeg){

Maamal.call(this,legs);

this.isVeg = isVeg;

}

Bat.prototype = Object.create(Maamal.prototype)

Bat.prototype.constructor = Bat;

Bat.prototype.fly = function(){

return'fly'

};

const fBat = new Bat(4,true)

console.log(fBat.sleep()) //sleepng

console.log(fBat.walk()) //walking

console.log(fBat.fly()) //fly

Object literal

# Object.setPrototypeOf() Method : Object Oriented Programming in JavaScript Series - Part 6

Using Object.setPrototypeOf() method to set one Object Literal's method to another.

Object.setPrototypeOf() Method JavaScript

There are two camps in JavaScript, one who uses function constructors to create objects and one who directly uses object literals. In JavaScript, function constructors can be extended by using the [Object.create()](http://techsith.com/object-create-method-javascript/" \t "_blank) method. This method lets you set [prototype](http://techsith.com/javascript-prototype-explained/) of one object to another object.

Constructor2.prototype = Object.create(Constructor1.prototype);

Extending Object Literals

Object literals don’t have prototype property. So, if you want to inherit properties of one literal to another, you can do it several ways. Prior to ES2015, you had to write your own extend method that would copy methods from one object to another. ES2015 introduced few new methods that makes this process easier. The setPrototypeOf() method is one of such methods.

Syntax

Object.setPrototypeOf(targetObj, sourceObj);

Here, targetObj is what you are setting prototype to from sourceObj. This method essentially sets a reference to sourceObj’s methods to targetObj’s \_\_proto\_\_. To understand the utility of this, let’s take two objects, toyota and camry. toyota has a drive() method. Setting the camry’s prototype to toyota makes toyota’s drive() method automatically available to camry. This is done by setting its reference in camry’s \_\_proto\_\_ property.

let toyota = {

drive() {

return 'driving toyota';

}

}

let camry = {

wifi() {

return 'using wifi';

}

}

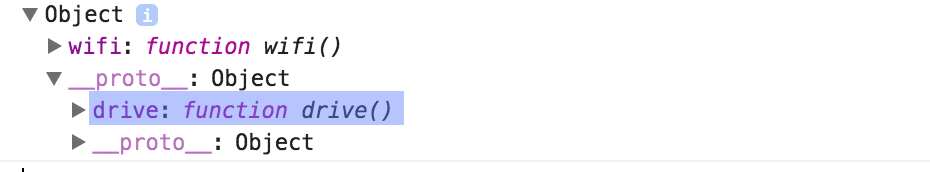
// Set toyota's \_\_proto\_\_ to camry's \_\_proto\_\_'s \_\_proto\_\_

Object.setPrototypeOf(camry, toyota);

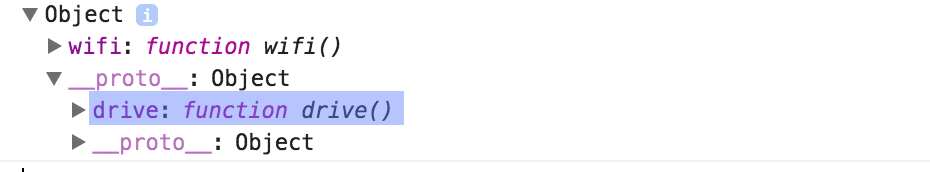
console.dir(camry); //prints the camry object

console.log(camry.wifi()); // using wifi

console.log(camry.drive()); // driving toyota

[](http://techsith.com/wp-content/uploads/2017/08/Screen-Shot-2017-08-06-at-12.37.27-AM.png)

If you inspect the camry object, you can view the \_\_proto\_\_ property and the reference to the drive() method of toyota. Since the setPrototypeOf() method sets a reference, any changes to toyota’s properties is automatically accessible to camry.

[](http://techsith.com/wp-content/uploads/2017/08/Screen-Shot-2017-08-06-at-12.37.27-AM.png)

If you inspect the camry object, you can view the \_\_proto\_\_ property and the reference to the drive() method of toyota. Since the setPrototypeOf() method sets a reference, any changes to toyota’s properties is automatically accessible to camry.

let toyota = {

drive() {

return 'driving toyota';

}

};

let camry = {

wifi() {

return 'camry';

}

};

// Set toyota's \_\_proto\_\_ to camry's \_\_proto\_\_'s \_\_proto\_\_

Object.setPrototypeOf(camry, toyota);

// Add a new Method to toyota object

toyota.newMethod = function() {

return 'new method from toyota';

};

console.log(camry.newMethod()); // Prints 'new method from toyota'

If you have a method with the same name in both the targetObject and the SourceObject, the method in the targetObject would have higher precedence. This is because javaScript first looks for object’s own method before looking into it’s prototype.

let toyota = {

drive() {

return 'driving toyota';

}

}

let camry = {

drive() {

return 'driving camry';

}

}

Object.setPrototypeOf(camry, toyota);

console.dir(camry.drive()); // logs "driving camry"

## Super keyword

You can call the method of the sourceObject from the targetObject’s method by using the super keyword.

let toyota = {

drive() {

return 'driving toyota';

}

}

let camry = {

drive() {

return `${super.drive()} camry`;

}

}

Object.setPrototypeOf(camry, toyota);

console.dir(camry.drive()); // logs "driving toyota camry"

## Avoid setting \_\_proto\_\_ manually

Prior to ES2015, Object.prototype.\_\_proto\_\_ was used to set the prototype of object literals. You can still use it as shown in the below example. However, it’s not recommended.

let toyota = {

print() {

return 'toyota';

}

}

let camry = {

printMe() {

return 'camry';

}

}

camry.\_\_proto\_\_ = toyota;

console.dir(camry.print());

# Object.assign() Method : Object Oriented Programming in JavaScript Series - Part 7

Use Object.assign() Method to copy methods from one object literal to another.

Instead of setting prototype link Use Object.assign() Method to copy methods from one object literal to another.

let toyota = {

drive() {

return 'driving toyota';

},

break(){

return 'breaking toyota'

}

}

let camry = {

wifi() {

return 'using wifi';

}

}

Object.assign(camry, toyota)

console.log(camry)

output:

* {
* break: break(){
* return 'breaking toyota'
* },
* drive: drive() {
* return 'driving toyota';
* },
* wifi: wifi() {
* return 'using wifi';
* }
* }

let toyota = {

drive() {

return 'driving toyota';

},

break(){

return 'breaking toyota'

}

}

let camry = {

drive() {

return 'driving camry ';

}

}

Object.assign(camry, toyota)

console.log(camry)

* {
* break: break(){
* return 'breaking toyota'
* },
* drive: drive() {
* return 'driving toyota';
* }
* }

Unlike setprototype of which keeps its own method here the methods are copied it will override the local method if we hv name collision

One of utility of Object.assign is used to create copy

let toyota = {

drive() {

return 'driving toyota';

},

break(){

return 'breaking toyota'

}

}

let camry = {

drive() {

return 'driving camry ';

}

}

let newCopyToyota = Object.assign({},toyota)

console.log(newCopyToyota)

* {
* break: break(){
* return 'breaking toyota'
* },
* drive: drive() {
* return 'driving toyota';
* }
* }

Copied all the methods it is doing Shallow copy which basically taking property and copy and it doesent do anything to prototype(if we have protype properties which will not copied)

We can make clone or add new properties to any object.

let toyota = {

drive() {

return 'driving toyota';

},

break(){

return 'breaking toyota'

}

}

Object.assign(toyota,{

wifi(){

return 'using wifi';

}

})

console.log(toyota)

* {
* break: break(){
* return 'breaking toyota'
* },
* drive: drive() {
* return 'driving toyota';
* },
* wifi: wifi(){
* return 'using wifi';
* }
* }

Utility of Object.assign If I have constructor

let c1 = function(x,y,z){

this.x = x;

this.y = y;

}

Instead of above method we can use

let c1 = function(x,y,z){

Object.assign(this,{x,y,z})

}

# Mixins in JavaScript : Object Oriented Programming Series - Part 8

# Mixins In JavaScript

### What is a Mixin?

Original concept of mixin came from an ice cream shop that provided on-demand customizable ice cream. At these shops, customers can make their own flavor by mixing up different toppings. You would pick up an ice cream flavor and a bunch of toppings, and then you can choose to mix them inside the ice cream by smashing these together instead of sprinkling as a topping. Hence, these are called mixins.

In terms of programming, Mixins are these pieces of code that you can mix with other objects to extend their functionality. Basically, you can borrow features from different mixins to compose the kind of object you desire

## Inheritance vs Composition

By using mixins, you can compose objects with a desired functionality. This is very different from using inheritance, which is much rigid and doesn’t work most of the time when you want to create dynamic ever-changing models. For example, you want to build a class called javaScriptEngineer, where javaScript means you can code in JavaScript and related technologies and Engineer means you have some sort of Engineering degree. Instead of using inheritance, where you would create an engineer class and extend it to make a JavaScript engineer, you can separate out both the functionalities into a separate mixins. You can compose javaScriptEngineer by using the Object.assign() method where you can borrow methods from two different mixins.

### Composing Object using Mixins

In the following example, we have two objects jsSkill and degree, they both would serve as mixins when I combine their properties and copy into an empty object using the Object.assign() method. The newly created object jsEngineer will have functionality of jsSkill and degree. Similarly, you can create a fullstackEngineer by using one more mixin. Since Fullstack engineer knows JavaScript, has an engineering degree, and also knows one or more backend languages, we just need to add an additional backendSkill mixin. This allows us to compose a fullstackEngineer.

const jsSkill = {

knowsJS() {

return true;

}

};

const degree = {

hasDegree() {

return true;

}

};

const backendSkill = {

knowsBackend() {

return true;

}

};

// Compose Objects

const jsProgrammer = Object.assingn({}, jsSkill);

const jsEngineer = Object.assign({}, jsSkill, degree);

const fullStackEngineer = Object.assign({}, jsSkill, backendSkill, degree);

console.log(jsEngineer.knowsJS());

console.log(jsEngineer.hasDegree());

### Factory Functions

Factory functions are functions that produces objects. In JavaScript, there are two ways to create objects, one by using either [function constructors](http://techsith.com/javascript-function-constructors/) or classes. In either case, you have to use a new keyword. And secondly, by using regular functions. When you use regular functions to build objects, they are called factory functions. Following example describes a factory function.

// Factory Function

let Car = function(color) {

return Object.assign({}, {

color: color

});

};

// New Object

let redCar = Car('red');

### Factory Function with closures

Having an inner variable inside the factory function will maintain the state inside a closure. The state will be stored inside every function that is using the variable.

// Factory Function

let Car = function(color) {

let isMoving = false;

return Object.assign({}, {

color: color,

drive(){

isMoving = true;

return this;

}, isMoving(){

return isMoving;

}

});

};

// New Object

let redCar = Car('red');

console.log(redCar.drive().isMoving())

### Function Mixins

Using a factory function with closures you can build function mixins. Each mixin would take an Object as a parameter and returns a new object that would take passed object and extend it with mixin’s functionality. Here is an example of building an object called superman using two mixins humanFactory and flyFactory. superman has characteristics of a human so he has an ability to cry, however unlike humans, he can fly. So, combining two different kind of abilities we can transform an ordinary reporter clark kent into the superman.

Note: When you pass an empty object to this factory, the returning Object would have a functionality of humanfactory, flyFactory and their preserved states.

/ fly factory function

const flyFactory = function(obj) {

// inner variable

let isFlying = false;

// returns a new object

return Object.assign({}, obj, {

fly() {

isFlying = true;

return this;

}, isFlying() {

return `${this.name} is ${(isFlying ? '' : 'not')} flying`;

}

});

};

// human factory function

const humanFactory = function(obj) {

let isCrying = false;

return Object.assign({}, obj, {

cry() {

isCrying = true;

return this;

}, isCrying() {

return `${this.name} is ${(isCrying ? '' : 'not')} crying`;

}

});

};

const clarkKent = {

name: 'clark kent'

};

// compose an object

const superman = humanFactory(flyFactory(clarkKent));

// set the state

superman.fly().cry();

console.log(superman.isFlying()); // prints: clark kent is flying

console.log(superman.isCrying()); // prints: clark kent is crying

### Conclusion

function mixin is one of the functional programming paradigm . Mixins help you reduce repetition of functions and promotes the reusability. This can reduce maintenance. It also support encapsulation using closures, with ability to inherit private data. In a way you can sort of achieve multiple inheritance since can inherit from multiple mixins. In this article, we discussed function mixins for a functional programming paradigm. The same concept also applies to ES2015 classes with a difference in syntax. To understand mixin for ES2015 classes, click here -> (class mixins coming soon)

# Javascript Classes ES2015 tutorial : Object Oriented Programming Series - Part 9

# JavaScript Classes

### Class Keyword

JavaScript uses the class keyword, which was introduced in ES2015. Prior to ES2015, you can create objects by using [function constructors](http://techsith.com/javascript-function-constructors/) and [prototype inheritance](http://techsith.com/javascript-prototype-explained/). There are many developers who believe that classes do not belong to JavaScript. They consider JavaScript as a functional programming language. I have heard opinions from the other side of the isle, where they believe that classes provide a less complicated solution. So regardless of an individual’s belief, classes are here! They are here to stay and are used extensively in modern frameworks like Angular, React, and Ember. Before we go deep into classes, let’s understand what does a class mean to JavaScript.

### Feels like Classes, Smells like Classes!

Introduction to classes in JavaScript was a great news for some, especially, former Java programmers who will feel like home. I believe, objective of people who added classes was just the same. However, I think a Java programmer will be more confused because under the hood classes do not act the same way as they do in Java. To quote a simple analogy, it’s like when you want your kids to eat more vegetables, you make a pizza, substitute tofu for cheese and spinach for pepperoni. Though you hide spinach in the middle layer, your kids can recognize that this is not a real pizza in their very first bite! Java programmers may feel the same as classes in JavaScript internally use the same function constructor and prototype methods from the versions prior to ES2015. Said that, syntactically classes are defined in the similar way as they are defined in Java.

### Syntax

In the example bellow, we have defined class Car. Notice the difference between the function declaration and class declaration? While declaring a class, we do not use semicolon. Similarly, to define a constructor, use the keyword ‘constructor’ and add the parameters to set the initial properties. If you check the type of the class you will be surprised that it’s actually “function” not “class”.

// class definition of Car

class Car {

constructor(color) {

this.color = color;

}

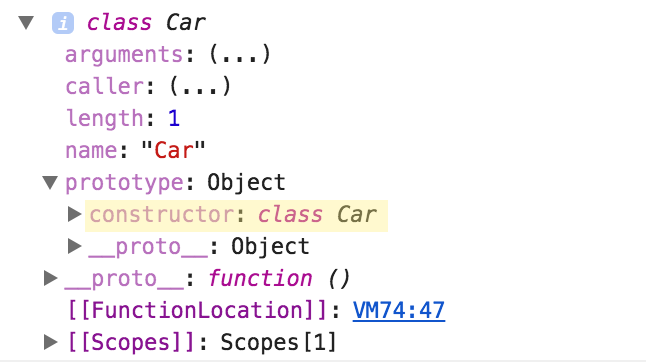
}

// Check the type of Car

console.log(typeof(Car)) //prints "function"

// inspecting

console.dir(Car);

[](http://techsith.com/wp-content/uploads/2017/08/Screen-Shot-2017-08-20-at-5.36.18-PM.png)Inspecting a JavaScript Class

If you inspect the Car object in detail you would see similar structure when you create object with function constructor. The only difference is that the constructor displays the keyword “class”.

## Simulating Class with Function Constructor

You can simulate the class behaviour using a function constructor wrapped inside an Immediately Invoked Function Expression (IIFE). This is obviously to see what is going on inside when you define a class.

// Simulating Class Car

let Car = (function(){

let Car = function(\_color){

this.color = \_color;

};

return Car;

})();

// Inspecting Car

console.dir(Car);

## Adding Methods

Now let’s add a drive() method to class Car. Note that there is no comma between two methods. It feels weird if all you have used is JSON Objects.

// Adding drive() method to Car

class Car {

constructor(color) {

this.color = color

}

drive() {

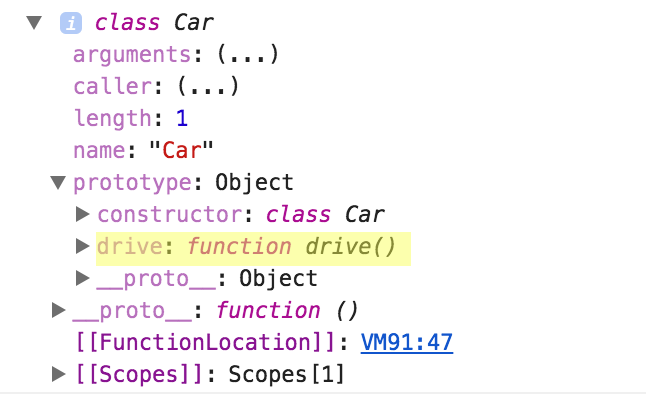
console.log('driving');

}

}

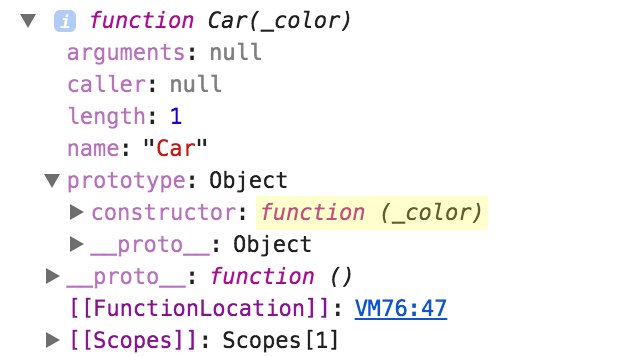
// Inspecting Car

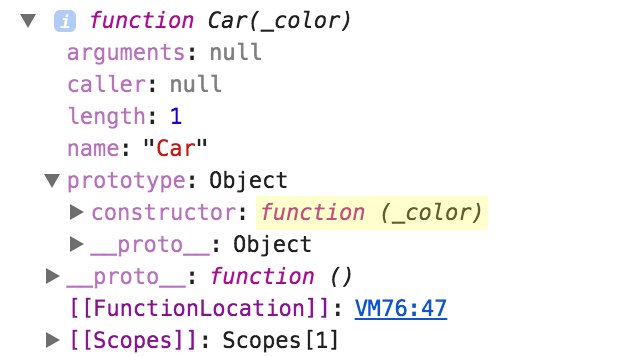
console.dir(Car);

[](http://techsith.com/wp-content/uploads/2017/08/Screen-Shot-2017-08-20-at-5.59.04-PM.png)Inspecting Class Method

When you inspect the class Car, you would notice the prototype property. Expanding prototype property displays the drive method. This means that Car is actually a function and drive is its prototype methods. Sounds familiar ? :). As I said, Class keyword is nothing but what some call “syntactic sugar”. All the methods are public. Unlike java, JavaScript doesn’t provide a way to add private methods.

Next, let’s build objects from class Car. The syntax is similar to what we used with function constructors, using the new keyword. You can execute the method on this object by calling redCar.drive().

[](http://techsith.com/wp-content/uploads/2017/08/Screen-Shot-2017-08-20-at-5.50.37-PM.png)Inspecting Function Constructor

[](http://techsith.com/wp-content/uploads/2017/08/Screen-Shot-2017-08-20-at-5.50.37-PM.png)Inspecting Function Constructor

When you inspect the class Car, you would notice the prototype property. Expanding prototype property displays the drive method. This means that Car is actually a function and drive is its prototype methods. Sounds familiar ? :). As I said, Class keyword is nothing but what some call “syntactic sugar”. All the methods are public. Unlike java, JavaScript doesn’t provide a way to add private methods.

Next, let’s build objects from class Car. The syntax is similar to what we used with function constructors, using the new keyword. You can execute the method on this object by calling redCar.drive().

// Adding drive() method to Car

class Car {

constructor(color) {

this.color = color

}

drive() {

console.log('driving');

}

}

// Creating an object using class Car

let redCar = new Car('red');

redCar.drive(); //Prints "driving"

# Javascript extend class es6 : Object Oriented Programming Series - Part 10

Classes:

It is a blueprint to create object it has property, constructor which is used to initialise property, method that would manipulate those property.

Subclasses: another class which inherit all the base class property and methods internally it uses inheritance.

class Mamal{

constructor(\_legs,\_name="xx"){

this.legs= \_legs;

this.name = \_name;

this.bool= true;

}

walk(){

return `'${this.name} is walking`

}

}

class Bat extends Mamal{

constructor(\_legs,\_name, \_eat){

super(\_legs,\_name);

this.eat = \_eat

}

fly(){

return `$this.name} is flying`

}

walk(){

let holding = this.eat? 'bug':'carrot'

return ` ${super.walk()}`

}

}

let fBat = new Bat(4,'peter',false)

console.log(fBat)

output : {

bool: true,

eat: false,

legs: 4,

name: "peter"

}

class Mamal{

constructor(\_legs,\_name="xx"){

this.legs= \_legs;

this.name = \_name;

this.bool= true;

}

walk(){

return `${this.name} is walking`

}

}

class Bat extends Mamal{

constructor(\_legs,\_name, \_eat){

super(\_legs,\_name);

this.eat = \_eat

}

fly(){

return `${this.name} is flying`

}

walk(){

let holding = this.eat? 'bug':'carrot'

return ` ${super.walk()}`

}

}

let fBat = new Bat(4,'peter',false)

console.log(fBat.fly()) // "peter is flying"

class Mamal{

constructor(a,b,c,d,){

Object.assign(this,{a,b,c,d})

}

}

class Bat extends Mamal{

constructor(f,...args){

super(...args);

this.f = f;

}

}const newBat = new Bat('f', 'a','b','c','d')

console.log(newBat)

{

a: "a",

b: "b",

c: "c",

d: "d",

f: "f"

}

# Static Methods in JavaScript Tutorial : Object Oriented Programming Series - Part 11

## 2 types of method : static and instance method

Static method: which will have static keyword Infront of method

Way to call is Class name.method name

Instance method are called on instance of a class(by using new kw)

Static method goes inside the class and instance method goes inside prototype

Class method or static are directly called on class itself

class Car{

static compare(){

console.log(this)

}

}

Car.compare()

Instance method

class Car{

static compare(){

console.log(this)

}

getColor(){

console.log(this)

}

}

const newcar= new Car()

newcar.getColor()

Car.compare()

Static method doesn’t use this to access the property, used as utility function

class Car{

constructor( color,price){

Object.assign(this,{color,price})

}

static compare(c1,c2){

return Math.abs(c1.price-c2.price)

}

getColor(){

return this.color

}

}

const red= new Car('red',100)

const blue= new Car('blue',102)

console.log(Car.compare(red,blue)) // 2

class Car{

constructor(p){

this.pp= p

}

static sell(c){

return `seeling for ${c.pp}`

}

}

class Toyota extends Car{

constructor(p){

super(p);

}

static sell(c){

return `Toyota ${super.sell(c)}`

}

}

const camy= new Toyota(100)

const blue= new Car('blue',102)

console.log(Toyota.sell(camy)) // "Toyota seeling for 100"

# Object Oriented Programming Series - Part 1

JavaScript Decorators ( Understanding Class and Property Descriptor Decorators )

Decorator: look more attracting by adding extra item

Class decorator:

Adding decorator to class using @with function name

let lipstic = function(target) {target.lips = 'pink';

}

@lipstic

class Girl {

}

console.log(`Her lips are ${Girl.lips}`)

output: "Her lips are pink"

if I want to have a choice of color, let’s say black

let lipstic = function(color) {

return function(target){

target.lips = color;

}

}

@lipstic('black')

class Girl {

}

console.log(`Her lips are ${Girl.lips}`)

output : "Her lips are black"

if I want to have another decorator

let lipstic = function(color) {

return function(target){

target.lips = color;

}

}

let earrings= function(target){

target.hasearrings = true;

}

@earrings

@lipstic('black')

class Girl {

}

console.log(`Her lips are ${Girl.lips} and she has earrings = ${Girl.hasearrings}`)

output : "Her lips are black and she has earrings = true"

lets say I have car class

class Car{

constructor(color){

this.color = color

}

getColor(){

return this.color;

}

}

const redCar= new Car('red');

console.log(redCar.getColor()) // "red"

in Js method can be overridden

redCar.getColor = function(){

return ‘blah’

}  
 console.log(redCar.getColor()) // " blah"

If I don’t want anybody to override this method after I define my class for that we can use decorator, but before I do that lets try to without decorator

There is an object method that is defineProperty which takes three argument

Which object I am defining this property to

Name of the method

configuration

class Car {

constructor(color) {

this.color = color

}

}

Object.defineProperty(Car.prototype, 'getColor', {

value: function() {

return this.color;

},

writable: false,

configurable:true,

enumerable: true

})

**const redCar = new Car('red');**

**redCar.getColor = function(){**

**return 'blah'**

**}**

**console.log(redCar.getColor())**

output : "red"

it has not overridden the property to blah

if I make writable : true it will override

if I want to make it clean,

class Car {

constructor(color) {

this.color = color

}

}

let descriptor = {

value: function() {

return this.color;

},

writable: true,

configurable: true,

enumerable: true

};

let readonly = function(target, key, descriptor) {

descriptor.writable = false;

return descriptor;

}

descriptor = readonly(Car.prototype, 'getColor', descriptor)

Object.defineProperty(Car.prototype, 'getColor', descriptor)

const redCar = new Car('red');

redCar.getColor = function() {

return 'blah'

}

console.log(redCar.getColor())

output : "red"

using decorator:

let readonly = function(target, key, descriptor) {

descriptor.writable = false;

return descriptor;

}

class Car {

constructor(color) {

this.color = color

}

@readonly

getColor(){

return this.color;

}

}

const redCar = new Car('red');

redCar.getColor = function() {

return 'blah'

}

console.log(redCar.getColor())

output : red

# Hoisting in JavaScript and function scope issues with keyword var

issues with keyword var

when you define a variable with var, that variable gets function scope(when I define a function and variable defined inside function is available throughout the function

var x = function(){

var y =1;

console.log(y) // 1

}

x()

var x = function(){

var y =1;

}

console.log(y) // y is not defined ref err

x()

y respects function boundary,

block scope: any block (open and close) : it should only exist between this bclock

var x = function(){

var y =1;

if(true){

let c = 2

console.log(c) // 2

}

console.log(c) //Uncaught ReferenceError

}

x()

c (using let )will be only accessible within if condition

using var inside if block

var x = function(){

var y =1;

if(true){

var c = 2

console.log(c) // 2

}

console.log(c) //2

}

x()

var does not respect if statement boundary

JS follows lexical scope, means y is within function x, function x has another function z, if I try to access y within z function it is accessible because y is declared with var it is a function scope

var x = function(){

var y =1;

var z = function(){

console.log(y) //1

}

z()

}

# 

problem with var keyword is hoisting:

hoisting:

var x = function(){

console.log(y) //undefined

console.log(c) //Uncaught ReferenceError

var y =1;

var z = function(){

console.log(y) //1

}

z()

}

x()

I have defined y variable below of console but if I m trying to access the viable before it is defining Js will not throw err but it gives undefined means it is till accessible, this is because whenever u define a variable within the body of function, the value will be available from that point to the end of the but before this variable it gives you y = undefined

What it essentially does is

var x = function(){

var y;

console.log(y) //undefined

y =1;

var z = function(){

console.log(y) //1

}

z()

}

x()

definition of y is hoisted to the top, as a programmer we are not expecting this one way to prevent it

always define the variable at the top of the function even if u using somewhere inside

var y = 2

var x = function(){

console.log(y) //undefined

var y =1;

}

x()

var y = 2

var x = function(){

console.log(y) // 1

}

x()

when you define a variable with var keyword, we can change the value any time, in programming language we cannot change the value once it is defined

var changes the value anytime, if you declare a variable with var keyword you can redefine the value any time

if you want to define constant in ES5you have to do like this

Object.definePropert(window, ‘PI’,{

Value:3.1.4,

Writable:false

})

But in es6 introced variable const, no need to write lengthy lines of code to make variable as constant

using constant we cannot change the value once it is defined

to solve all the problem with var , JavaScript introduced let keyword and const keyword

var:function scope

let: block scope

usng const

const pi = 3.14;

pi = 2 // err, assignment to const varible

Let:

Let keyword solve the problem of var

Which gives block scope

It doesn’t do hoisting

let x = function(){

if(true){

let y= 1;

console.log(y) //

}

console.log(y) //"Uncaught ReferenceError: y is not defined"

}

x()

if I try to access variable outside the block it won’t be available,

if I try to do before the assignment of I , it will give not defined

let x = function(){

if(true){

console.log(y) //"Uncaught ReferenceError: Cannot access 'y' before initialization"

let y= 1;

}

}

# Javascript Immediately Invoked Function Expression IIFE - Tutorial

//function declaration

function increment(i){

return i+1;

}

//function expression

var increment = function(i){

return i+1

};

increment(i)

we can call this function anytime using increment(i)(we can invoke at some later stage)

there are times when u want to execute function right away as soon as u define it , and u will not gonna call again

if you define and execute in the same sentence : IF WE not gonna invoke again we don’t need function declaration and function expression

you can invoke it using IIFE

(function(i){

return i+1;

})(j)

This function would be immediately invoked and argument passing j is passes I, it also anonymous fun there is no name

Other way of writing is

(function(i){

return i+1;

}(j))

If you are not returning any value back you can also do like this: valid syntax

!function(){

//code here

}()

-function(){

//code here

}()

+function(){

//code here

}()

~function(){

//code here

}()

Usage:

Minifying : we can minify window.JQuery) to $ reducing scope look up,bit faster, takes global object and makes it local

function($){

$(this).addClass('MyClass')

})(window.JQuery)

Ex of creating own JS library,

Here is a IIFE fun exp, it execute right away, whatever it returns it copied into counter, inside fun private variable I = 0; returning the object with 3 method these are private method, this is also closure becse it is a private variable it is not exposed outside,if I try to use variable outside the function I would not be able to gives err,because it is a clousure , it holds the variable inside, it is persisting the variable inside

var counter = (function(){

var i=0 ;

return {

get:function(){

return i;

},

set: function(val){

i = val;

},

increment: function(){

i++

}

}

})()

console.log(counter.get()) //0

counter.set(5)// 5

counter.increment()

console.log(counter.get()) //6

# Javascript Closure tutorial ( Closures Explained )

"Closures are nothing but FUNCTIONS WITH PRESERVED DATA",

So what is closure in javascript? "whenever you declare a function inside another function, the inside function(s) is/are recreated each time the outside function is called"

"Closures are functions that refer to independent (free) variables. In other words, the function defined in the closure 'remembers' the environment in which it was created."

Usually when we create fun either u pass some parameter or define some inner variable

Here I have function and passing passed, and inner variable : in this case returns 5;

var addTo = function(passed){

var inner = 2;

return passed + inner;

}

console.log(addTo(2)) //4

but in JS you can get by without passing any variables:

you can define the var outside and if I run this I get the same ans

var passed = 2

var addTo = function(){

var inner = 2;

return passed + inner;

}

console.log(addTo()) //4

this is a closure

in js variable are defined outside the function is automatically available inside the function since JS follows lexical scoping

variable defined inside the function is not accessible outside but anything defined outside the function is accessible inside the function

how do it doing ? it is doing with closure

any fun which is using outside variable is closure .

var passed = 3;

var addTo = function () {

var inner = 2;

return passed + inner;

};

console.log(addTo());//5

var passed = 4;

console.log(addTo());//6

lets see another ex, it is similar to previous example but it is broken into 2 fun

var addTo = function(passed){

var add = function(inner){

return passed+ inner;

}

return add;

}

let addThree = new addTo(3)

let addToFour = new addTo(4)

console.log(addThree) // returns add function

console.log(addToFour) // returns add function

console.log(addThree(1)) // 4

console.log(addToFour(1)) //5

we can create unlimited no of function as many as I can using clossure

it keeps the variable it needed to execute, it preserve the variables inside the function as a property closure , when u execute add three it uses that closure to the calculation

closures are the function with preserved data

Closures in JavaScript | Inside a loop, inner function and setTimeoout

Why do we use closure?

JS has lexical scoping (outside variable are automatically available inside the function)

**Problem 1**:

local variables are created when the function is called and they will be destroyed when the function task is finished. It means local variables have shorter life time than global variables. We may use global variables to overcome that issue.

Global variables are available when the program starts and when it ends. They are also available throughout the program.

**Problem 2**:

Since global variables are accessible throughout the program, they are prone to change from everywhere.

**What do we want?**

We want to have data persistence + data encapsulation.

We can achieve them by using Closures. By using a closure we can have private variables that are available even after a function task is finished.

let i = 1;

const fun = ()=>{

console.log(i) // 1 becz of lexical scope i is available inside the function

}

fun()

we are trying to use the variable outside the function what if the variable is not available

what if I is not available when we execute the function , what if I move this function somewhere else and trying to access it in js or call it using set timeout

when we define a function in 1 scope and execute in another scope outside variable is not available

if(true){

var i = 1;

const fun = ()=>{

console.log(i) // ReferenceError: fun is not defined

}

}

fun()

in the above ex I is inside a block and we are calling the function outside(I will not be available outside the block), the fun defined inside scope using let kw is not available once the scope is done

to solve the problem

define the function name outside then inside provide definition of the function

let fun;

if(true){

var i = 1;

fun = ()=>{

console.log(i) // 1

}

}

console.log(i) // I will not be available

fun() // I will be avilable

console.log(fun()) // this function will holds the I in scopes, (holding I inside block)

define your function somewhere and execute it somewhere else, closure will help

let f =()=>{

let i = 1;

let j= 2;

return ()=>{

console.log(i)

}

}

console.log(f())

when I call function f it return anonymous function ()=>{

console.log(i)

}

it will have closure only for i (it doesn’t close j inside it )because I is used inside the function, j is not used inside the function

closure inside the loop

for(let i=0;i<3;i++){

const f =()=>{

console.log(i). ///0,1,2

}

f()

}

closure in settimeout with let keywoed

for(let i=0;i<3;i++){

setTimeout(()=>{

console.log(i)

},1000)

}

Output //0,1,2 with let keyword

When I =0; f(c:i) becse I am using let this is a block scope as soon as the for loop ends, a new for loop comes back in I doesn’t exist it would give me new I, which is 1, the function will hold recent i

I = 0 f(c:i(0));

I = 1 f(c:i(1))

I=2 f(c:i(2))

I would get 3 diff function which holds 3 separate i

Suppose if I change let to var

for(var i=0;i<3;i++){

setTimeout(()=>{

console.log(i)

},1000)

}

Output: 3,3,3

Because var has a function scope it doesn’t really create new variable it just changes the I variable value this is not the new variable for the 2nd iteration it is a same variable with different value second function still holds the closure as the original I, but now the value has changes

I = 0 f(c:i(2))

I = 1 f(c:i(2))

I = 2 f(c:i(2))

In order to verify it less than 3

It should got to 3 and then it stops, it would increment one more level, I is now 3, technically it is same I, because of the same I it updated all the value to 3(because of the value changes it updates the closure)

Lets say I don’t want to use let and I want to keep it as var

If I want to keep it as var how do fix it?

To fix this issue, I need to wrap it in a function and immediately run it

for (var i = 0; i < 3; i++) {

((i) => {

setTimeout(() => {

console.log(i)

}, 1000)

})(i)

}

Output: 0,1,2

Reason why it is working is because Immediately invoked function creates a block scope because it runs right away

**javascript callback functions tutorial(higher order functions)**

**in js functions are 1st class object**

**Passing function as an argument into function**

**let x = function(){**

**console.log("i am called from inside function")**

**}**

**let y = function(callback){**

**console.log("do something")**

**callback()**

**}**

**Output: do something, i am called from inside function")**

**y(x)function x is passing to function y as function body and we can execute it some point**

**why do we need call-back function**

**without callback**

**let calck = function(n1, n2, type) {**

**if (type == 'add') {**

**return n1 + n2**

**} else if (type == "mul") {**

**return n1 \* n2**

**}**

**}**

**console.log(calck(2, 3, 'add')) // 5**

**using callback**

**let add = function(a, b) {**

**return a + b;**

**}**

**let mul = function(a, b) {**

**return a \* b;**

**}**

**let doWhatever = function(a, b) {**

**return console.log("nothing a,b")**

**}**

**let calck = function(n1, n2, callback) {**

**return callback(n1, n2)**

**}**

**console.log(calck(2, 3, add)) //5**

**as a user I can create any function**

**I can write function directly as an argument , I don’t have to define the function outside**

**let cal = function(n1, n2, callback) {**

**return callback(n1, n2)**

**}**

**console.log(cal(2, 3, function(a, b) {**

**return a - b**

**}))**

**When you want to use only once we can do above method**

**When I pass cab function and execute it I need to make sure that user is passing function not something else**

**let cal = function(n1, n2, callback) {**

**if (typeof callback === 'function') {**

**return callback(n1, n2)**

**} else{**

**consol.log('pas')**

**}**

**}**

**console.log(cal(2, 3, function(a, b) {**

**return a - b**

**}))**

# JavaScript Practical Applications of Call, Apply and Bind functions

let add = function(c){

console.log( this.a + this.b +c )

}

let obj = {

a:1,

b: 2

}

add.call(obj,3) //6

you can call add function on the object obj using add.call(obj,3), I have used add function on this object. Add is not part of this object but we can use it using call,

every object in JS has prototype, within the prototype we will find bunch of method,

for ex if I take array, it has proto inside it, it has bunch of method

Utilities

let argsToArray = function(){

console.log(arguments) //[object Arguments] {  
 0: 1,  
 1: 2,  
 2: 3  
}

console.log( [].slice.call(arguments) ) //[1, 2, 3]

}

argsToArray(1,2,3)

here the argsToArray is a function, in JS you don’t have to pass an argument because inside there is an argument object available , it’s called s argument, wich holds all the parameter passed to the function,

[object Arguments] {  
 0: 1,  
 1: 2,  
 2: 3  
}

argument object automatically available inside ,if I look at the arguments it should give me object with 3 element, it is not an array, its an array like object, if I look at it it doesn’t have proto

I want to convert tis to an array, I can use arrays functionality to convert it to array am gonna borrow functionality from arrays prototype

The way I should do is

[].slice.call(arguments) which will convert argument to an array

We are borrowing slice functionality from array to convert argument into an array

You can borrow array functionality on non array object

2 utility:

When you are not using classes u r using function constructor and u have base constructor and sub constructor, if I want to call base constructor from sub constructor instead of using super() we can use call()

The way it work is

let mammal = function(legs){

this.legs= legs;

}

let cat = function(legs,isDomesticatd){

mammal.call(this,legs);

this.isDomesticatd = isDomesticatd;

}

let lion = new cat(4, false)

console.log(lion)

here mammal is a base constructor and cat is a sub constructor or sub class, inside of sub classs I don’t have to set yup legs again. I take mammal call on this

lion is an instance of the constructor

cat{  
 isDomesticatd: false,  
 legs: 4  
}

unlike a call where u pass argument apply would take arrays and argument, the utility of apply function would be when you pass an array as an argument it converts into regular arguments

let nuArray = [1,2,3];

console.log(Math.min(1,2,3)) //1

console.log(Math.min.apply(null,nuArray)) //1

we cannot pass nuArray to Math.min function, we have to pass argument to function,

using call we can pass nuArray

I can make nuArray as list of argument, I cann pass this into call method

Math.min.apply(null,nuArray) // nuArray gets converted into argument

BIND: just like call and apply, we can borrow functionality from other libraries on ur object, bind will bind will bind the the borrowed functionality and give you functionality bind inside ur object

let button = function(content) {

this. content = content

}

button.prototype.click = function() {

console.log(`${this.content } clicked`) // "undefined clicked"

}

let newButton = new button('add')

let looseClick = newButton.click()

looseClick();

here the click function is loosely bound to new button because it is no longer bound, we can verify by doing this

let looseClick = newButton.click()

I can bound it by calling bound boutton

let button = function(content) {

this.content = content

}

button.prototype.click = function() {

console.log(`${this.content} clicked`) // "add clicked"

}

let newButton = new button('add')

let boundClick = newButton.click().bind(newButton)

boundClick ()

we can bind back using bind()

let myObj = {

asyncGet(cb){

cb()

},

parse(){

console.log('parse called') //"Uncaught TypeError: this.par

},

render(){

this.asyncGet(function(){

this.parse()

})

}

}

myObj.render()

because this inside async function means I am calling scope of this is asyncget ,

one solution is to make that = this

let myObj = {

asyncGet(cb){

cb()

},

parse(){

console.log('parse called') //"parse called"

},

render(){

that = this;

this.asyncGet(function(){

that.parse()

})

}

}

myObj.render()

but it is not good idea instead I hv to do is

let myObj = {

asyncGet(cb){

cb()

},

parse(){

console.log('parse called') //" "parse called"

},

render(){

that = this;

this.asyncGet(function(){

that.parse()

}.bind(this))

}

}

myObj.render()

basically it is binding outer this to inner this

## **When should you NOT use arrow functions in ES6? Name three or more cases.**

**Senior**

[](https://www.fullstack.cafe/interview-questions/javascript" \o "JavaScript Interview Questions)**[JavaScript](https://www.fullstack.cafe/interview-questions/javascript" \o "JavaScript Interview Questions)**[142](https://www.fullstack.cafe/interview-questions/javascript" \o "JavaScript Interview Questions)

Answer

Arrow functions should NOT be used:

* When we want function hoisting - as arrow functions are anonymous.
* When we want to use this/arguments in a function - as arrow functions do not have this/arguments of their own, they depend upon their outer context.
* When we want to use named function - as arrow functions are anonymous.
* When we want to use function as a constructor - as arrow functions do not have their own this.
* When we want to add function as a property in object literal and use object in it - as we can not access this (which should be object itself).

# Async Await JavaScript ES7

ASYNC: You need to wait, JavaScript is non-blocking by default it doesn’t wait for async

console.log('person1 shows ticket');

console.log('person2 shows ticket');

const preMovie = async () => {

const person3PromiseToShowTicketWhenWifeArrives = new Promise((resolve, reject) => {

setTimeout(() => resolve('ticket'), 3000);

});

const getPopcorn = new Promise((resolve, reject) => {

setTimeout(() => resolve('popcorn'), 3000);

});

const addButter = new Promise((resolve, reject) => {

setTimeout(() => resolve('butter'), 3000);

});

let ticket = await person3PromiseToShowTicketWhenWifeArrives;

console.log(`got the ${ticket}`);

console.log(`Husband:we should go in now`);

console.log(`Wife: "i am hungry"`);

let popcorn = await getPopcorn;

console.log(`Husband: here is ${popcorn}`);

console.log(`Husband:we should go in now`);

console.log(`Wife: "I dont like popcorn without butter!"`);

let butter = await addButter;

console.log(`added ${butter}`);

console.log(`Husband:Anything else darling`);

console.log(`Wife: lets go we are going to miss the preivew`);

console.log(`Husband: thanks for the reminder \*grin\*`);

return ticket;

};

preMovie().then((t) => console.log(`person4 shows ${t}`));

console.log('person4 shows ticket');

output

"person1 shows ticket"

"person2 shows ticket"

"person4 shows ticket"

"got the ticket"

"Husband:we should go in now"

"Wife: \&quot;i am hungry\&quot;"

"Husband: here is popcorn"

"Husband:we should go in now"

"Wife: \&quot;I dont like popcorn without butter!\&quot;"

"added butter"

"Husband:Anything else darling"

"Wife: lets go we are going to miss the preivew"

"Husband: thanks for the reminder \*grin\*"

"person4 shows ticket"

All the staents executes only after one another, I cannot really [ut them in between

Instead of this mess I will use wait

console.log('person1 shows ticket');

console.log('person2 shows ticket');

const preMovie = async () => {

const person3PromiseToShowTicketWhenWifeArrives = new Promise((resolve, reject) => {

setTimeout(() => resolve('ticket'), 3000);

});

const getPopcorn = new Promise((resolve, reject) => {

setTimeout(() => resolve('popcorn'), 3000);

});

const getCandy = new Promise((resolve, reject) => {

setTimeout(() => resolve('candy'), 3000);

});

const getCoke = new Promise((resolve, reject) => {

setTimeout(() => resolve('coke'), 3000);

});

let ticket = await person3PromiseToShowTicketWhenWifeArrives;

let [ popcorn, candy, coke ] =

await Promise.all([ getPopcorn, getCandy, getCoke ]);

console.log(`got ${popcorn}, ${candy}, ${coke}`);

return ticket;

};

preMovie().then((t) => console.log(`person4 shows ${t}`));

console.log('person4 shows ticket');

output :

person1 shows ticket

person2 shows ticket

person4 shows ticket

got popcorn, candy, coke

person4 shows ticket

await can be only used inside async body, if I pul any console.log after awit it will not execute until await function executes

usually in the promise resolve reject we can hav try catch to handle the exceptional case

# Applications of JavaScript Spread Operators Tutorial | ES6 / ES2015

Let’s say if u want to create duplicate array

let x = [1,2,3,4]

I wanna create copy of it

Let y= x;

This is not a copy what it actually do is, reference of x is set to y, if I change x, y also will change

But we want to create copy of x

let x = [1,2,3,4]

let y = Object.assign([],x)

console.log(y) //[1, 2, 3, 4]

the new array which is a copy of x, it is a shallow copy because it just copies a elememt not any prototypeanther way to do is

let x = [1,2,3,4]

let y = [...x]

console.log(y) //[1, 2, 3, 4]

here

…x = 1,2,3,4

[…x] = [1,2,3,4]

If I put [] around … it becomes array

The same way we can merge 2 array

let x = [1,2,3,4]

let y = [5,6,7]

let z = [...x,..y]

console.log(z) // [1,2,3,4,5,6,7]

here ...x = 1,2,3,4;

...y = 5,6,7;

I can also do in this way

let x = [1,2,3,4]

let y = [5,6,7]

console.log(x.concat(...y)) //[1, 2, 3, 4, 5, 6, 7]

another use of spread operator is array to argument

let x = [1,2,3]

console.log(...x) // 1,2,3

let func = function(...arg){

console.log(arg) //[1, 2, 3]

}

func(...x)

func(...x) would become like func(1,2,3)

in older days befor spread operator we used to do something like this

let x = [1,2,3]

let func = function(...arg){

console.log(arg) //[1, 2, 3]

}

func.apply(null,x)

because u can convert array to argument, whichever function takes argument u can pass as an array

for ex:

in the math library we have function called hypot , what it does is, whatever the parameter we pass to this function, it will do square of each no and add them and makes square root of the number

console.log(Math.hypot(3,4)) //5

instead of this I can simply make an array and pass it to the hypot function(if I have lots of number I have to pass it an argument instead of passing all the number to a function I can make array and using spread opertoar I can pass it

let x= [3,4]

console.log(Math.hypot(...x)) // 5

let x = [1,2,4,2,4,5,5,2,6,7,9]

Math.min(..x) //1

# Javascript Default Parameters ES6 ES2015 tutorial

let add =(a,b)=>{

return a+b

}

console.log(add()) // NAN

this is because we need to handle this type of saturation where user doesn’t provide any parameter ,

usually what we do is

let add =(a,b)=>{

if(a === undefined){

a= 0;

}

if(b=== undefined){

b= 0

}

return a+b;

}

console.log(add()) //0

the better way to handle this situation is

let add = (a, b) => {

a = a || 0;

b = b || 0;

return a + b

}

console.log(add()) //0

if I provide only 1 parameter that means I ma providing a,

let add = (a, b) => {

a = a || 0;

b = b || 0;

return a + b

}

console.log(add(1)) //1

if I want to provide value for b

let add = (a, b) => {

a = a || 0;

b = b || 0;

return a + b

}

console.log(add(undefined, 1)) //1

we can do more concise syntax

let add = (a, b) => {

return (a || 0) + (b||0);

}

console.log(add()) //1

in ES6 we can do this

let add = (a=0,b=0) => {

return a+b;

}

console.log(add()) //0

console.log(add(undefined,1)) //1

console.log(add(1,1)) //2

I ca say b= a , whatever the value of a I am assigning it to b

let add = (a=0,b=a) => {

return a+b;

}

console.log(add()) //0

console.log(add(undefined,1)) //1

console.log(add(1)) //2

if I am passing an an array (if I am adding 2 array)

let add = (a=[],b=[]) => {

return [...a,...b];

}

console.log(add()) // []

console.log(add([1])) // [1]

console.log(add([1], [2])) // [1,2]

insead of pasing this type of value, what id I am pasing an object

let x = {a:1,b:2}

let add = (obj) => {

return obj.a+ obj.b;

}

console.log(add(x)) // 3

console.log(add()) // "Uncaught TypeError: Cannot read property

to solve the uncaught type err issue

let x = {a:1,b:2}

let add = (obj) => {

obj = obj || {}

let a = obj.a || 0;

let b = obj.b || 0;

return a+b

}

console.log(add(x)) // 3

console.log(add()) // 0

this is till long thing to do

instead of object, I can put definition of object

let x = {a:1,b:2}

let add = ({a=0,b=0}={}) => {

return a+b

}

console.log(add(x)) // 3

console.log(add()) // 0

# javaScript Method chaining tutorial ( function chaining)

var obj = function(){

this.i = 0;

this.add = function(i){

this.i += i;

};

this.substract = function(i){

this.i -=i;

};

this.print = function(){

console.log(this.i) //1

};

}

var x = new obj();

x.add(3)

x.substract(2);

x.print()

which works fine, but this is not function chaining,

I want to do,

x.add(3). Substract(2).print()

in the above ex if I do

console.log(x.add(3)) I will get undefined,if I have undefined I cannot apply x.add(3). Substract(2).print()

solution to that is

if I return the entire object

var obj = function(){

this.i = 0;

this.add = function(i){

this.i += i;

return this;

};

this.substract = function(i){

this.i -=i;

return this;

};

this.print = function(){

console.log(this.i) //1

};

}

var x = new obj();

x.add(3).substract(2).print()

if I have closure and I want to do same thing,

in order to create closure I need to have private properties and variable

I cannot really use it like above becaze these are private method it is still ot accessible outside, I am going to return the object

I don’t have to use keyword new

var obj = function(){

var i = 0;

var add = function(j){

i += j;

return this;

};

var substract = function(j){

i -=j;

return this;

};

var print = function(){

console.log(i) //1

};

return {

add:add,

substract:substract,

print:print

}

}

var x = obj();

x.add(3).substract(2).print()

I have 1, I have simply passed all the function as object, I still don’t have I , because I is a closure

Each function inside the object has closure for I, that’s how we do function chaining in closure

# JavaScript Sets and Weaksets | ES2015

JavaScript Data Structure Set and Weaksest. Sets lets you store unique values of any type. We understand its syntax, use and some cool applications.

In JS we have 2 data type:

1. Primitive: single value, non-mutable, string number boolean
2. Reference : array, object

Sets are reference data type,

Sets allows used to store unique values of any data type, means we cannot have duplicate values inside a set, it basically wont allow, it is unique list

To compare with array

const ary = []; this synatax is called array literal

I can also define array like this using constructure,

const myArray = new Array();

when we comes to set we have to use constructor, it doesn’t have literal sysntax

const mySet = new Set();

console.log(mySet)

*Set(0) {}*

*[[Entries]]*

*No properties*

size: (...)

\_\_proto\_\_: Set

add: *ƒ add()*

clear: *ƒ clear()*

constructor: *ƒ Set()*

delete: *ƒ delete()*

entries: *ƒ entries()*

forEach: *ƒ forEach()*

has: *ƒ has()*

keys: *ƒ values()*

size: (...)

values: *ƒ values()*

Symbol(Symbol.iterator): *ƒ values()*

Symbol(Symbol.toStringTag): "Set"

get size: *ƒ size()*

\_\_proto\_\_: Object

Right now it doesn’t have value, so entries are empty here

We cannot access set as u access array

Ex, for array we can add like this

Const array = []

Crray[0] =1

I cannot do like this in set, Set makes use of prototype method

To add value into set I have to to s=use add method

const mySet = new Set()

mySet.add(1)

mySet.add(2)

console.log(mySet) //

*Set(2) {1, 2}*

*[[Entries]]*

size: (...)

\_\_proto\_\_: Set

add: *ƒ add()*

clear: *ƒ clear()*

constructor: *ƒ Set()*

delete: *ƒ delete()*

entries: *ƒ entries()*

arguments: (...)

caller: (...)

length: 0

name: "entries"

\_\_proto\_\_: *ƒ ()*

*[[Scopes]]*: Scopes[0]

forEach: *ƒ forEach()*

has: *ƒ has()*

keys: *ƒ values()*

size: (...)

values: *ƒ values()*

Symbol(Symbol.iterator): *ƒ values()*

Symbol(Symbol.toStringTag): "Set"

get size: *ƒ size()*

\_\_proto\_\_: Object

As u see above it doesn’t have [ ] like array it has { } like object literal

When I run a method on set it returns me Set back, I can use function chaining

mySet.add(1).add(2).delete(1)

*et(1) {2}*

*[[Entries]]*

0: 2

size: (...)

\_\_proto\_\_: Set

add: *ƒ add()*

clear: *ƒ clear()*

constructor: *ƒ Set()*

delete: *ƒ delete()*

entries: *ƒ entries()*

forEach: *ƒ forEach()*

has: *ƒ has()*

keys: *ƒ values()*

size: (...)

values: *ƒ values()*

Symbol(Symbol.iterator): *ƒ values()*

Symbol(Symbol.toStringTag): "Set"

get size: *ƒ size()*

\_\_proto\_\_: Object

Since I deleted 1 only 2 left,

I can clear the entire set using clear(), its gonna empty the set

mySet.add(1).add(2).delete(1).clear()

array have length, sets have size

mySet.size

I don’t always have to add it like this , I can create array, and convert it to set if I want to

const array = [1,2,3,4]

const mySets = new Set(array)

console.log(mySets)

*Set(4) {1, 2, 3, 4}*

*[[Entries]]*

0: 1

1: 2

2: 3

3: 4

size: (...)

it would actually convert array into set,

set only takes an argument something that is itereable(array) it wont take JS object

const mySet. = new Set([1,2,3,4])

mySet.add(1).add(2)

console.log(mySet)

// 1,2,3,4

Since 1, and 2 are already there, if I try to add it will not add into to the set, since characteristics of sets are it only holds unique value, sets are iterable onjct like array,

We can use for of loop

const mySet = new Set([1,2,3,4])

for(let val of mySet){

console.log(val)

}

mySet.add(2)

output

1

2

3

4

All sets are not array because there is no orderning, they are also not key value pair like object they are something of its own u=you could iterate them like array, ie is because it has entries whicj holds it like array, that’s wht we can iterate it,

Sets can be useful when u have 2 sets, u want to interest them and union them,

It has delete method for it to create union and intersect it, if you want to access the elemet with index then array ara better option

Set allows to remove duplicate value form an array

If I concert array into set I would have set with elemet 1, 2,3 its not ana rray, my object is to get the array back I can do it using spread operator

const a = [1,2,3,4,3]

console.log([... new Set(a)]) // [1, 2, 3, 4]

I can alos use, Array.from to conert set into an array

const a = [1,2,3,4,3]

console.log(Array.from(new Set(a))) // [1, 2, 3, 4]

Weak Sets:

Weak Sets are also constructor but instead of set we use weak set, it laso has add mwethod, unlike sets u can add primitive value here u can only add objects,

U can alos pass an array to weak set with values as object, no primitive values

const ws = new WeakSet()

ws.add({a:1})

console.log(ws)

*WeakSet {{…}}*

*[[Entries]]*

*No properties*

\_\_proto\_\_: WeakSet

add: *ƒ add()*

constructor: *ƒ WeakSet()*

delete: *ƒ delete()*

has: *ƒ has()*

Symbol(Symbol.toStringTag): "WeakSet"

\_\_proto\_\_: Object

const ws = new WeakSet([{a:1},{b:2}])

console.log(ws)

*WeakSet {{…}, {…}}*

*[[Entries]]*

*No properties*

\_\_proto\_\_: WeakSet

add: *ƒ add()*

constructor: *ƒ WeakSet()*

delete: *ƒ delete()*

has: *ƒ has()*

Symbol(Symbol.toStringTag): "WeakSet"

\_\_proto\_\_: Object

I can pass an iterable object however weak set are not iterable ,

const ws = new WeakSet([{a:1},{b:2}])

for(let val of ws){

console.log(val) // ws is not iterable

}

Application: lewants say I have class, u can acces the method directly, if u wanna make sure that user has call method check if it has reference to that object,

const we = new WeakSet()

class SomeClass {

constructor(){

ws.add(this)

}

method(){

if(!ws.has(this)){

throw new TypeEroor(' SomeClass prototype method called incompatible object}

# JavaScript Maps & WeakMaps

ES6 Introduced new Data Structures Map and WeakMap in JavaScript. Understand difference between JavaScript objects and Maps

Maps allows map arbitory value to other value,

Problem with regular object

I want to use key as object

const x = {};

const a = {};

const b = {

num: 1

}

x[a] = 'a'

console.log(x)

console.log(x[a]) // ‘’a’’

*{[object Object]: "a"}*

[object Object]: "a"

\_\_proto\_\_: Object

I would get object object as the key and a as the value which is fine

Lets add another key

const x = {};

const a = {};

const b = {

num: 1

}

x[a] = 'a'

x[b] = 'b'

console.log(x)

1. *{[object Object]: "b"}*
   1. [object Object]: "b"
   2. \_\_proto\_\_: Object

Here I have only one key object object , js object only supports 1 key object, if I use multiple key , then it only remembers the problem,

Hence we need map

const map = new Map()

console.log(map)

1. *Map(0) {}*
   1. *[[Entries]]*
      1. *No properties*
   2. size: (...)
   3. \_\_proto\_\_: Map
      1. clear: *ƒ clear()*
      2. constructor: *ƒ Map()*
      3. delete: *ƒ delete()*
      4. entries: *ƒ entries()*
      5. forEach: *ƒ forEach()*
      6. get: *ƒ ()*
      7. has: *ƒ has()*
      8. keys: *ƒ keys()*
      9. set: *ƒ ()*
      10. size: (...)
      11. values: *ƒ values()*
      12. Symbol(Symbol.iterator): *ƒ entries()*
      13. Symbol(Symbol.toStringTag): "Map"
      14. get size: *ƒ size()*
      15. \_\_proto\_\_: Object

It has iterator which is gret because JS object doesn’t iterate

const x = {};

const a = {};

const b = {

num: 1

}

const map = new Map()

map.set(a, 'a')

map.set(b, 'b')

console.log(map)

1. *Map(2) {{…} => "a", {…} => "b"}*
   1. *[[Entries]]*
      1. 0: {Object => "a"}
         1. key: {}
         2. value: "a"
      2. 1: {Object => "b"}
         1. key:
            1. num: 1
            2. \_\_proto\_\_: Object
         2. value: "b"
   2. size: (...)
   3. \_\_proto\_\_: Map

It has 2 keys, one is empty object and second is num: a

We can do chining also like

map.set(a,’a’).set(b,’b’)

what if I want to same key again with different value

const x = {};

const a = {};

const b = {

num: 1

}

const map = new Map()

map.set(a, 'a').set(b,'b').set(a,'c')

console.log(map)

1. *Map(2) {{…} => "c", {…} => "b"}*
   1. *[[Entries]]*
      1. 0: {Object => "c"}
      2. 1: {Object => "b"}
   2. size: (...)
   3. \_\_proto\_\_: Map

in map it only store the unique value, it really overrides the 1st one it only remembers the last one

if iwant to remove b

const x = {};

const a = {};

const b = {

num: 1

}

const map = new Map()

map.set(a, 'a').set(b,'b').set(a,'c')

map.delete(b)

console.log(map)

1. *Map(1) {{…} => "c"}*
   1. *[[Entries]]*
      1. 0: {Object => "c"}
   2. size: (...)
   3. \_\_proto\_\_: Map

Unlike regular js object if I want to iterate value I need to use for in loop which is bad, here I can use for of loop, because I hav iterator property, this allows to iterate through map

const x = {};

const a = {};

const b = {

num: 1

}

const map = new Map()

map.set(a, 'a').set(b, 'b').set(a, 'c')

for (let [key, value] of map.entries()) {

console.log(key, value)

}

Output :

{} "c"

{num: 1} "b"

Because of this I can also convertmap into an array

const x = {};

const a = {};

const b = {

num: 1

}

const map = new Map()

map.set(a, 'a').set(b, 'b').set(a, 'c')

const ary = [...map]

console.log(ary)

1. *(2) [Array(2), Array(2)]*
   1. 0: (2) [{…}, "c"]
   2. 1: (2) [{…}, "b"]
   3. length: 2
   4. \_\_proto\_\_: Array(0)
2. *(2) [Array(2), Array(2)]*
   1. 0: Array(2)
      1. 0: {}
      2. 1: "c"
      3. length: 2
      4. \_\_proto\_\_: Array(0)
   2. 1: Array(2)
      1. 0: {num: 1}
      2. 1: "b"
      3. length: 2
      4. \_\_proto\_\_: Array(0)
   3. length: 2
   4. \_\_proto\_\_: Array(0)

It converts into 2 dimensional array

Why do we need weak map”?

Once we define map key, it holds it in memory, it doesn’t let garbage collected even the original key is not removed that’s why we need weak map

{

let x = {

a: [1, 2]

}

var map = new Map();

map.set(x, 'something')

}

console.log(map)

1. *Map(1) {{…} => "something"}*
   1. *[[Entries]]*
      1. 0: {Object => "something"}
         1. key:
            1. a: (2) [1, 2]
            2. \_\_proto\_\_: Object
         2. value: "something"
   2. size: (...)
   3. \_\_proto\_\_: Map

We have x is key let has block scope but var has function scope it should exist outside,

We already know that x doesn’t exist outside, it should not be available outside the block.

However we are getting key here because it has attached to the map , it has. Not garbage collected, even though x doesn’t exist, still key is patrt odf thre map

If I use weak map instead of map

{

let x = {

a: [1, 2]

}

var map = new WeakMap();

map.set(x, 'something')

}

console.log(map)

1. *WeakMap(1) {{…} => "something"}*
   1. *[[Entries]]: Array(0)*
   2. Length:0

If you are not refreshing the code it will work but If u just refresh this code it will not work

Answer to you question: x was not garbage collected , it will stays there until console refreshed or you forcefully delete it from map

# Javascript pass by Value vs pass by Reference tutorial

Does JavaScript use pass by reference or pass by value when i pass a variable to a function ( Primitive or Object

If u pass a variable into a function using pass by reference then if u change the value of the variable inside the function the change gets reflected outside

If u pass by value then it actually makes a copy and into it, if u change the value inside, it doesn’t get reflected outside

let a = 1;

let change = (val)=>{

val = 2

}

change(a)

console.log(a) //1

in js always pass by reference

when u pass a primitive value, it is pas by value\when you pass object it is pass by reference but that’s not true, in js it is always pass by reference

let a = {

num: 1

}

let change = (val) => {

val = {}

}

change(a)

console.log(a)

output:

{num: 1}

In the above ex:

Here a is object it has num :1, inside I am emptying the object,

When I pass a inside it reassigned it to empty object, it is actually pass by reference instead of replacing entire object I m going to mutate it:

let a = {

num: 1

}

let change = (val) => {

val.num = 2;

}

change(a)

console.log(a)

output:

{num: 2}

Which means it is pass by reference, and that is why when I mutate it that mutation is also it reflected in a,

In js when u assign value to a variable it actually holds the address of the variable where it get stored, so it has reference to the object, when I pass a to the function, it actually passing the reference but when I use = it means I am reassigning the value it actually changes the object, as if like it recreating the same variable again, because this scope is different it actually creates a variable val inside it when I reassign it, val is empty object and a is completely different object num=1

let a = {

num: 1

}

let b = a;

a.num = 2 ;

console.log(a) //2

console.log(b) //2

let a = {

num: 1

}

let b = a;

a.num = 2 ;

a = {};

console.log(a) // {}

console.log(b) //2

if I reassign a = {} I would get different result, reassignment actually loose the reference, when you mutate the object it keeps its reference because of the same object

, but in primitive type we cannot mutate it

# stack implementation in JavaScript

when u execute the js all the function inside js gets pushed on stack and browser executed from the stack, reverse string,

// create a stack in JavaScript

function CreateStack(maxSize = 10) {

const \_stack = [];

let \_top = -1;

// Add on top

function push(val) {

if (\_top >= maxSize - 1) {

return "stack overflow";

} else {

\_stack.push(val);

\_top++;

}

}

// remove from top

function pop() {

if (\_top > -1) {

\_top--;

return \_stack.pop();

} else {

return null;

}

}

// get lenght

function size() {

return \_top + 1;

}

// get the top value

function peek() {

if (\_top > -1) {

return \_stack[\_top];

} else {

return null;

}

}

// print as string

function print() {

console.log(\_stack.toString());

}

// find value

function search(val) {

const temp = [];

let found = null;

while (\_top > -1) {

if (\_stack[\_top] === val) {

found = \_top;

break;

}

temp.push(\_stack.pop());

\_top--;

}

const templen = temp.length;

for (let i = 0; i < templen; i++) {

\_stack.push(temp.pop());

}

return found;

}

// empty the stack

function empty() {

while (\_top > -1) {

\_stack.pop();

\_top--;

}

}

return { push, print, pop, size, peek, search, empty };

}

const myStack = CreateStack(1000);

myStack.push("hi");

myStack.push("hello");

myStack.pop();

myStack.print();

# Big O Notation | time complexity of algorithms

Big “O” notation allows us to compare the worst case performance of our algorithm in a standardised way

Sum = n(n+1)/2 =🡺O(1)

# Debounce in Javascript

Imagine, u have shopping app, when user purchase the item and clicks twice on submit button, what happens, the order will get submitted twice?

Debounce: is used to prevent lots of events fired.

Debounce means: if u keep clicking on the same button multiple times, it will execute the even @the last on last click.

<button id=”myid>Click me</button

const debounce =(fn,delay)=>{

let timeoutId;

return function(...args){

if(timeoutId){

clearTimeout(timeoutId)

}

timeoutId = setTimeout(()=>{

fn(...args)

},delay)

}

}

document.getElementById('myid').addEventListener(

"click",

debounce(e => {

console.log('clicked');

},2000)

)

# Throttling in JavaScript

They both avoid unnecessary action to be fired

In debounce last clicks get fired after the delay

In throttling, first click gets fired

<button id=”myid>Click me</button

const throttle = (fn,delay)=>{

let last = 0

return (...args)=>{

const now = new Date().getTime();

if(now-last<delay){

return;

}

last = now;

return fn(...args);

}

}

document.getElementById('myid').addEventListener(

'click',

throttle( ()=> {

console.log('you clicked me')

},5000))

Initially when u click the button the events gets fired and after that when u keeps on clicking then after 5 seconds the event will get triggered

**click me**

"you clicked me" when time is 0sec

"you clicked me" // after 5 sec

# The Differences Between forEach() and map() that Every Developer Should Know

Map method receives function as a parameter, then it applies on each element and returns entirely new array populated with the result of calling provided function

This means that it returns a new array that contains an image of each element of the array, it will always return the same number of items

**const myArray = [1,2,3,4];**

**myArray.map(x=>x\*x)**

**(4) [1, 4, 9, 16]**

forEach() method receives a function as an argument and executes it once for each array element. However , instead of returning new array like map,it returns undefined

const myArray = [1,2,3,4];

myArray.forEach(x=>x\*x)

undefined

Difference:

The returning value

The 1st difference between map() and forEach() is the returning value. The forEach() method returns undefined and map() returns a new array with the transformed elements. Even if they do the same job, the returning value remains different.

Ability to chain other methods

Map() is chainable.this means that we can attach reduce(),sort(),filter() and so onafter performing map method on an array.

That’s something we cant do with forEach() because, as might guess it returns undefined

const myAwesomeArray = [1, 2, 3, 4, 5]

myAwesomeArray.forEach(x => x \* x).reduce((total, value) => total + value)

//>>>>>>>>>>>>> Uncaught TypeError: Cannot read property 'reduce' of undefined

myAwesomeArray.map(x => x \* x).reduce((total, value) => total + value)

//>>>>>>>>>>>>>return value: 55

1. Mutability

The map() method returns an entirely new array with transformed elements and the same amount of data , in the case of foreach(), even if it is returns undefined, it will mutate the original array with the callback

# The JavaScript Reduce Method Explained

The signature for the reduce array method in JavaScript is:

arr.reduce(callback,initialValue)

Reduce comes with some terminology such as **reducer & accumulator**. The accumulator is the value that we end with and the reducer is what action we will perform in order to get to **one value**.

You must remember that a **reducer** will only return one value and one value only hence the name **reduce**.

Take the following classic example:

const value = 0;

const numbers = [5, 10, 15];

for(let i = 0; i < numbers.length; i++) {

value += numbers[i];

}

The above will give us 30 (5 + 10 + 15). This works just fine, but we can do this with reduce instead which will save us from mutating our value variable.

The below code will also output 30, but will not mutate our value variable (which we have now called initialValue)

/\* this is our initial value i.e. the starting point\*/

const initialValue = 0;

/\* numbers array \*/

const numbers = [5, 10, 15];

/\* reducer method that takes in the accumulator and next item \*/

const reducer = (accumulator, item) => {

return accumulator + item;

};

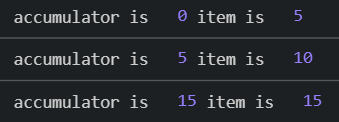
/\* we give the reduce method our reducer function

and our initial value \*/

const total = numbers.reduce(reducer, initialValue)

 Let’s add a console.log in our reducer method that will output the accumulator and the item arguments.

The following screenshot shows what’s logged to the console:



So the first thing we notice is our method is called 3 times because there are 3 values in our array. Our accumulator begins at 0 which is our initialValue we passed to reduce. On each call to the function the item is added to the accumulator. The final call to the method has the accumulator value of 15 and item is 15, 15 + 15 gives us 30 which is our final value. Remember the reducer method returns the accumulator plus the item.

## Final Example - Changing an Object Structure

So with the new Pokemon game coming out, let’s pretend we have a server that sends us an array of Pokemon objects like so:

const pokemon = [

{ name: "charmander", type: "fire" },

{ name: "squirtle", type: "water" },

{ name: "bulbasaur", type: "grass" }

]

Copy

We want to change this object to look like:

const pokemonModified = {

charmander: { type: "fire" },

squirtle: { type: "water" },

bulbasaur: { type: "grass" }

};

Copy

To get to that desired output we do the following:

const getMapFromArray = data =>

data.reduce((acc, item) => {

// add object key to our object i.e. charmander: { type: 'water' }

acc[item.name] = { type: item.type };

return acc;

}, {});

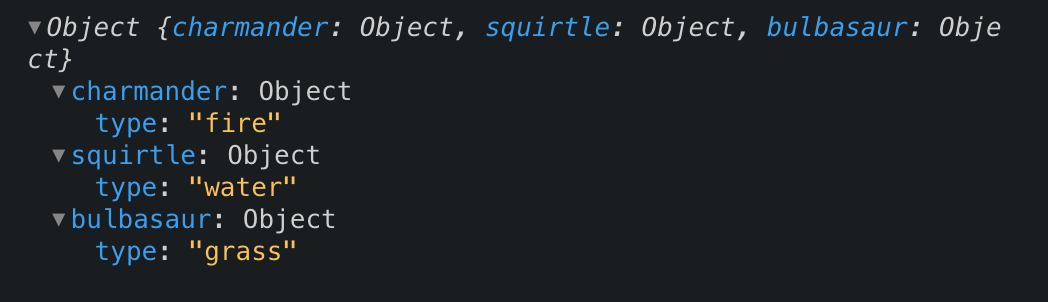
Copy

If we call our method like so:

getMapFromArray(pokemon)

Copy

We get our desired output:



# Array.from Method in JavaScript

Array has prototype method, called from,

[1,2,3].slice()

Array.from(): it does not directly work on array, it works on array constructor method.

Let str = “techslith”

I want to convert this into an array

Console.log([…str]) // [“t”,”e”,”c”,”s”,”l”,”t”,h”]

Or I can do like this

Const fromString = Array.from(str) //// [“t”,”e”,”c”,”s”,”l”,”t”,h”]

Or

I can also create array from from

I can create empty array of certain size, 1st argument would be object and second argument is call back function

const fromScratch = Array.from({length:5},(v,i)=>v);

console.log(fromScratch)//[undefined, undefined, undefined, undefined, undefined]

const fromScratch = Array.from({length:5},(v,i)=>i);

console.log(fromScratch)//[0, 1, 2, 3, 4]

const fromScratch = Array.from({length:5},(v,i)=>i\*i);

console.log(fromScratch)//[0, 1, 4, 9, 16]

without using Array.from() how can u do it

const fromScratch = new Array(5)

console.log(fromScratch) //[undefined, undefined, undefined, undefined, undefined]

const fromScratch = (new Array(5)).fill(0)

console.log(fromScratch) //[0, 0, 0, 0, 0]

const fromScratch = new Array(5).fill(0).map((v,i)=>i)

console.log(fromScratch) //[0, 1, 2, 3, 4]

if I want to get the unique value from an array

cons ary = [1,1,2,3]

const unique = Array.fome(new Set(ary)

console.log(unique) ))//[1,2,3]

I I don’t want to use Array.from

Const unique = […new Set(ary)]

Console.log(unique) //))//[1,2,3]

Another utility is

const func = function(){

console.log(arguments)

}

func()

1. *Arguments [callee: ƒ, Symbol(Symbol.iterator): ƒ]*
   1. callee: *ƒ ()*
   2. length: 0
   3. Symbol(Symbol.iterator): *ƒ values()*
   4. \_\_proto\_\_: Object

const func = function(){

console.log(arguments)

}

func('t','e','c','s','i','t','h')

Arguments(7) ["t", "e", "c", "s", "i", "t", "h", ]

const func = function(){

console.log(Array.from(arguments))

}

func('t','e','c','s','i','t','h') //["t", "e", "c", "s", "i", "t", "h"]

const func = function(){

console.log(Array.from(arguments).join(''))

}

func('t','e','c','s','i','t','h') //tecsith

or

const func = function(...args) {

console.log(args.join(''));

}

func('t','e','c','s','i','t','h')

# JavaScript Array Methods , flat , flatMap , reduceRight, copyWithin

**flat**

const matrix =[[1,2,3],[4,5,6]]

const flatArray = matrix.flat(1) //need to provide how deep I want to flattens it

console.log(flatArray)//[1, 2, 3, 4, 5, 6]

what if I have very deep matrix

const matrix =[[1,2,3],[[[[[[[5]]]]]]]]

const flatArray = matrix.flat(Infinity)

console.log(flatArray)//[1, 2, 3, 4, 5]

if u have multi dimensional array and u want to add all the elements in the array then

const matrix =[[1,2,3],[[[[[[[5]]]]]]]]

const flatArray = matrix.flat(Infinity).reduce((a,v)=>a+v,0)

console.log(flatArray)//11

**flatMap**

const nums= [1,2,3];

const strs = ['one','two','three']

const mapped = nums.map((val,index)=>[val,strs[index]])

console.log(mapped)

output:it gives 3 array

1. *(3) [Array(2), Array(2), Array(2)]*
   1. 0: (2) [1, "one"]
   2. 1: (2) [2, "two"]
   3. 2: (2) [3, "three"]
   4. length: 3
   5. \_\_proto\_\_: Array(0)
2. *(3) [Array(2), Array(2), Array(2)]*
   1. 0: (2) [1, "one"]
   2. 1: (2) [2, "two"]
   3. 2: (2) [3, "three"]
   4. length: 3
   5. \_\_proto\_\_: Array(0)

The way I wanted is single dimensional array(note: it works only for 1D array)

const nums= [1,2,3];

const strs = ['one','two','three']

const mapped = nums.flatMap((val,index)=>[val,strs[index]])

console.log(mapped)

output:

[1, "one", 2, "two", 3, "three"]

**reduceRight**

**reduce: usually takes from left to right**

const str = ['t','s','e','b'];

let best = str.reduce((c,v)=>c+v);

console.log(best)//tseb

**reduceRight: takes value from right to left**

const str = ['t','s','e','b'];

let best = str.reduceRight((c,v)=>c+v);

console.log(best)//best

reduce and reduceRight produce same result for number because for adding no order is not matter

**copyWithin**

**copy some of the elements within the array**

const nums = [0,1,2,3,4,5,6,7,8,9];

nums.copyWithin(0)

console.log(nums)//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

const nums = [0,1,2,3,4,5,6,7,8,9];

nums.copyWithin(1)// which means which index I am copying,it mutates the arrray

console.log(nums)//[0, 0, 1, 2, 3, 4, 5, 6, 7, 8]

const nums = [0,1,2,3,4,5,6,7,8,9];

nums.copyWithin(3)

console.log(nums)// [0, 1, 2, 0, 1, 2, 3, 4, 5, 6]

const nums = [0,1,2,3,4,5,6,7,8,9];

nums.copyWithin(0,2)

console.log(nums)// [2, 3, 4, 5, 6, 7, 8, 9, 8, 9]

const nums = [0,1,2,3,4,5,6,7,8,9];

nums.copyWithin(0,3)

console.log(nums)// [3, 4, 5, 6, 7, 8, 9, 7, 8, 9]

# JavaScript ES2020 awesome new features.

* + 1. globalThis:

console.log(globalThis)

console.log(self)

console.log(frames)

console.log(this)

these are reference to the window object,

if u r usin g webworker only delf would work , js wanted to have common global and that worlks acrsoos all platform and diff places, they created globalThis

Window {parent: global, opener: null, top: global, length: 0, frames: Window, …}

1. *Window {parent: global, opener: null, top: global, length: 0, frames: Window, …}*
   1. alert: *ƒ alert()*
   2. atob: *ƒ atob()*
   3. blur: *ƒ blur()*
   4. btoa: *ƒ btoa()*
   5. caches: CacheStorage {}
   6. cancelAnimationFrame: *ƒ cancelAnimationFrame()*
   7. cancelIdleCallback: *ƒ cancelIdleCallback()*
   8. captureEvents: *ƒ captureEvents()*
   9. chrome: {app: {…}, runtime: {…}, loadTimes: *ƒ*, csi: *ƒ*}
   10. clearInterval: *ƒ clearInterval()*
   11. clearTimeout: *ƒ clearTimeout()*
   12. clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor: "Google Inc.", maxTouchPoints: 0, hardwareConcurrency: 16, …}
   13. close: *ƒ close()*
   14. closed: false
   15. confirm: *ƒ confirm()*
   16. createImageBitmap: *ƒ createImageBitmap()*
   17. crypto: Crypto {subtle: SubtleCrypto}
   18. customElements: CustomElementRegistry {}
   19. defaultStatus: ""
   20. defaultstatus: ""
   21. devicePixelRatio: 2
   22. document: document
   23. external: External {}
   24. fetch: *ƒ fetch()*
   25. find: *ƒ find()*
   26. focus: *ƒ focus()*
   27. frameElement: null
   28. frames: Window {parent: global, opener: null, top: global, length: 0, frames: Window, …}
   29. getComputedStyle: *ƒ getComputedStyle()*
   30. getSelection: *ƒ getSelection()*
   31. history: History {length: 10, scrollRestoration: "auto", state: null}
   32. indexedDB: IDBFactory {}
   33. innerHeight: 369
   34. innerWidth: 244
   35. isSecureContext: true
   36. length: 0
   37. localStorage: (...)
   38. location: Location {href: "https://fiddle.jshell.net/\_display/?editor\_console=true", ancestorOrigins: DOMStringList, origin: "https://fiddle.jshell.net", protocol: "https:", host: "fiddle.jshell.net", …}
   39. locationbar: BarProp {visible: true}
   40. matchMedia: *ƒ matchMedia()*
   41. menubar: BarProp {visible: true}
   42. moveBy: *ƒ moveBy()*
   43. moveTo: *ƒ moveTo()*
   44. name: "result"
   45. navigator: Navigator {vendorSub: "", productSub: "20030107", vendor: "Google Inc.", maxTouchPoints: 0, hardwareConcurrency: 16, …}
   46. onabort: null
   47. onafterprint: null
   48. onanimationend: null
   49. onanimationiteration: null
   50. onanimationstart: null
   51. onappinstalled: null
   52. onauxclick: null
   53. onbeforeinstallprompt: null
   54. onbeforeprint: null
   55. onbeforeunload: null
   56. onblur: null
   57. oncancel: null
   58. oncanplay: null
   59. oncanplaythrough: null
   60. onchange: null
   61. onclick: null
   62. onclose: null
   63. oncontextmenu: null
   64. oncuechange: null
   65. ondblclick: null
   66. ondevicemotion: null
   67. ondeviceorientation: null
   68. ondeviceorientationabsolute: null
   69. ondrag: null
   70. ondragend: null
   71. ondragenter: null
   72. ondragleave: null
   73. ondragover: null
   74. ondragstart: null
   75. ondrop: null
   76. ondurationchange: null
   77. onemptied: null
   78. onended: null
   79. onerror: *(message, url, line, column) => {…}*
   80. onfocus: null
   81. onformdata: null
   82. ongotpointercapture: null
   83. onhashchange: null
   84. oninput: null
   85. oninvalid: null
   86. onkeydown: null
   87. onkeypress: null
   88. onkeyup: null
   89. onlanguagechange: null
   90. onload: null
   91. onloadeddata: null
   92. onloadedmetadata: null
   93. onloadstart: null
   94. onlostpointercapture: null
   95. onmessage: null
   96. onmessageerror: null
   97. onmousedown: null
   98. onmouseenter: null
   99. onmouseleave: null
   100. onmousemove: null
   101. onmouseout: null
   102. onmouseover: null
   103. onmouseup: null
   104. onmousewheel: null
   105. onoffline: null
   106. ononline: null
   107. onpagehide: null
   108. onpageshow: null
   109. onpause: null
   110. onplay: null
   111. onplaying: null
   112. onpointercancel: null
   113. onpointerdown: null
   114. onpointerenter: null
   115. onpointerleave: null
   116. onpointermove: null
   117. onpointerout: null
   118. onpointerover: null
   119. onpointerrawupdate: null
   120. onpointerup: null
   121. onpopstate: null
   122. onprogress: null
   123. onratechange: null
   124. onrejectionhandled: null
   125. onreset: null
   126. onresize: null
   127. onscroll: null
   128. onsearch: null
   129. onseeked: null
   130. onseeking: null
   131. onselect: null
   132. onselectionchange: null
   133. onselectstart: null
   134. onstalled: null
   135. onstorage: null
   136. onsubmit: null
   137. onsuspend: null
   138. ontimeupdate: null
   139. ontoggle: null
   140. ontransitionend: null
   141. onunhandledrejection: null
   142. onunload: null
   143. onvolumechange: null
   144. onwaiting: null
   145. onwebkitanimationend: null
   146. onwebkitanimationiteration: null
   147. onwebkitanimationstart: null
   148. onwebkittransitionend: null
   149. onwheel: null
   150. open: *ƒ open()*
   151. openDatabase: *ƒ ()*
   152. opener: null
   153. origin: "https://fiddle.jshell.net"
   154. outerHeight: 878
   155. outerWidth: 1418
   156. pageXOffset: 0
   157. pageYOffset: 0
   158. parent: global {window: global, self: global, location: {…}, closed: false, frames: global, …}
   159. performance: Performance {timeOrigin: 1603019904046.489, onresourcetimingbufferfull: null, memory: MemoryInfo, navigation: PerformanceNavigation, timing: PerformanceTiming, …}
   160. personalbar: BarProp {visible: true}
   161. postMessage: *ƒ postMessage()*
   162. print: *ƒ print()*
   163. prompt: *ƒ prompt()*
   164. queueMicrotask: *ƒ queueMicrotask()*
   165. releaseEvents: *ƒ releaseEvents()*
   166. requestAnimationFrame: *ƒ requestAnimationFrame()*
   167. requestIdleCallback: *ƒ requestIdleCallback()*
   168. resizeBy: *ƒ resizeBy()*
   169. resizeTo: *ƒ resizeTo()*
   170. screen: Screen {availWidth: 1792, availHeight: 1030, width: 1792, height: 1120, colorDepth: 30, …}
   171. screenLeft: 121
   172. screenTop: 118
   173. screenX: 121
   174. screenY: 118
   175. scroll: *ƒ scroll()*

ANGULAR

BehaviourSubject: this would helps us update back and forth

Angular 4/5 , using service and reactive extension (rxjs) share/sync data across components. So data updates to one component would get automatically to all the components sharing that service.

Creating a user that we are going to share

Private user= new BehaviourSubject<string>(‘john’);

cast = this.user.asObservable(); I can observe the change from anywhere

editUser(newUser){

this.user.next(newUser) /// replaced the value of user

}

1. Promise.allSettled()

Allows us to find out if all the promise that u want to Execute have been settled

const p1 = new Promise((resolve,reject)=>setTimeout(resolve,2000))

const p2 = new Promise((resolve,reject)=>setTimeout(reject,3000))

const p3 = new Promise((resolve,reject)=>setTimeout(resolve,4000))

Promise.allSettled([p1,p2,p3]).then(res=>res.forEach(result=>console.log(result)));

Promise.allSettled(): either it can be resolved or reject that wait until all three are done(to be either reolved or rejected, but it gives me the sstus when all three are done

Promise.all(): they all 3 have to be resolved , as soon as the 1st promise got rejected it will give an err

1. *{status: "fulfilled", value: undefined}*
   1. status: "fulfilled"
   2. value: undefined
   3. \_\_proto\_\_: Object

console.js:127

1. *{status: "rejected", reason: undefined}*
   1. reason: undefined
   2. status: "rejected"
   3. \_\_proto\_\_: Object

console.js:127

1. *{status: "fulfilled", value: undefined}*
   1. status: "fulfilled"
   2. value: undefined
   3. \_\_proto\_\_: Object

3. Nullish Coalescing Operator(??)

This is actually ??

let x = {

profile: {

age: 27,

name: "john"

}

}

console.log(x.profile.name); //”john”

what if name doesn’t exist

let x = {

profile: {

age: 27,

}

}

console.log(x.profile.name); //undefined

if there is no name there is an operator to provide default value

let x = {

profile: {

age: 27,

}

}

console.log(x.profile.name || ‘john’); //undefined

double pipe: above code: if the name or age is undefined then give the values as john, what it does is it checks empty value, the problem with this is what if I have empty string

let x = {

profile: {

age: 0,

name: ""

}

}

console.log(x.profile.name || 'john');//"john"

console.log(x.profile.age || 21);//21

it thinks that epty strib is empty value, empty string is an empty string it si not an empty value, it shoud give empty tring not john same thing for age, it should show 0 but age is a valid no it is not undefined, to solve this problem es2020 introduce ??

this is the case I want to print the actual value

let x = {

profile: {

age: 0,

name: ""

}

}

console.log(x.profile.name ? ? 'john') //''

console.log(x.profile.age ? ? 21); // 0

let x = {

profile: {

}

}

console.log(x.profile.name ? ? 'john') //"john"

console.log(x.profile.age ? ? 21); // 25

* 1. Optional Chaining Operator (?..)

let x = {

profile: {

name: ""

}

}

console.log(x.profile.age) //undefined

but what if I don’t have even profile

let x = {

}

console.log(x.profile.age) //"Uncaught TypeError: Cannot read property 'age' of undefined"

if it have 2 level of undefined the it would give an err, which could be an issue

let x = {

}

console.log(x && x.profile && x.profile.name)//undefined

this would give me an undefined

the problem with above syntax, I need to write condition for object, object property, jS introduced new operator to solve this issue

let x = {

}

console.log(x?.profile?.name); //undefined

* 1. BigInt

const max = Number.MAX\_SAFE\_INTEGER

console.log(max); //9007199254740991

the length of no is 16 what if I have no bigger than this and I want to use it in js

console.log(max ); //9007199254740991

console.log(max +1 ); //9007199254740992

console.log(max +2); //9007199254740992

it gives me the same value, it is weired, sometimes it gives me exact value and somtime it is giving unpredictable value

bigint helps to do that

const bigOne = 1n;

console.log(bigOne + 1n)//2n

console.log(bigOne + 2n) //3n

if u want to concert from another data type to big intger u have to use constructor

const bigOne = BigInt('1')

console.log(bigOne)//1n

const bigOne = BigInt(false)

console.log(bigOne)//0n

# Angular 5 Unit Testing with Jasmine Karma and Protractor

Protactor:

Helps to run test against the browser,  
k=jasin:behiviour driven test env

Karma:

Run the test in multiple browser

You write ur test in jasmin, it will run in protractor, it will allow u to do in multiple browser using karma

# Angular 5 setup with SASS, LESS preprocessor tutorial

Ng new ng5-sass –style=scss

If I want to convert project from sass to less

Ng new ng5-less –style= less

Instead of above cmd

“styleEXt=”less” in the angular.json file

Ng set defaults.styelExt less

“stules”:[

“less/styles.less”]

WHAT IS PROTOTYPE INHERITANC IN JS

Basically every object in js has a property called protoptype, where u can add methods and propertis to it, and when u create other object from this object the newly created object will automatically inherit the property of the parent but not by including in its own property but instead it uses fromits parent prototype properties and methods

The way it works is when u call a method on the object it 1st llooks at its own property to see if it is there, if it is not there then it will look at its parent and if it finds it then it will execute so this way the objects are much lighter and doesn’t carry all this method with it

Graphical user interface, text, application, chat or text message

Description automatically generated

What is promises and why do we use it?

When u want to make async call which has to wait for something to happen and once it comes back you exist with the callback function , within that call back function u might do another ajax call which can wait for another result and u would ave another callabck function on success or failure and it can become kind of nested call bck function hell

To simpliy this thing we use promises

Closure and how do u use it

When a function returns another function the returning function will hold its enviornemnet basically all the variable is needed, its environment

Graphical user interface, text, application, chat or text message

Description automatically generated