**Java script is an Object Oriented language**

Java script frames are below:

* **Angular**
* **React**
* **Ember**

To execute Java Script, we need only browser (any supported browser no external setup is required)

Console.log (“Hello World”);

Console – global object, log – Method

JavaScript Variable Declaration:

var value = 42, var is Keyword

var value = hello;

var value;

Value =42;

Java script does not have type of variable like (int, string, float, double)

In java script—

No need to declare variable type

The same variable can be assigned values of different types

No Scoping information on variable declaration (ex: private, public)

JavaScript provides different data types to hold different types of values. There are two types of data types in JavaScript:

1. **Primitive values**

Non-primitive values (object references)

The fundamental difference between primitives and non-primitives is that primitives are immutable and non-primitives are mutable.

Primitive types:

* **Number**
* **String**
* **Boolean**
* **Undefined**
* **null**

Primitives are known as being immutable data types because there is no way to change a primitive value once it is created

**Example**:

var String =” This is a string”;

String [1] – output is ‘h’, if you try to change the value of ‘h’ to some other letter ‘F’

String [1] = ‘F’

console.log(string) // 'This is a string.' (No change in the first letter, which has changed to ‘F’)

Primitives are compared by value. Two values are strictly equal if they have the same value.

var number1 = 5;  
var number2 = 5;

number1 === number 2; // true

var string1 = 'This is a string.';  
var string2 = 'This is a string.';

string1 === string2; // true

1. **Non-primitive types:**

* Functions
* Objects
* Arrays

The above are referred to as non-primitive values.

Non-primitive values are mutable data types. The value of an object can be changed after it gets created.

**Example**:

var arr = [ 'one', 'two', 'three', 'four', 'five' ];

arr[1] = 'TWO';

console.log(arr) // output is [ 'one', 'TWO', 'three', 'four', 'five' ];(we can change the value of arr like this)

Objects are not compared by value. This means that even if two objects have the same properties and values, they are not strictly equal. Same goes for arrays. Even if they have the same elements that are in the same order, they are not strictly equal.

**Example**:

var obj3 = { 'car' : 'purple' }  
var obj4 = obj3;

obj3 === obj4; // true

**Example**:

var obj1 = { 'cat': 'playful' };  
var obj2 = { 'cat': 'playful' };

obj1 === obj2; // false

var arr1 = [ 1, 2, 3, 4, 5 ];  
var arr2 = [ 1, 2, 3, 4, 5 ];

arr1 === arr2; // false

Note: Non primitive values can also be referred to as reference types(address) because they are being compared by reference instead of value. Two objects are only strictly equal if they refer to the same underlying object.

Apart from primitive data type there are two more type introduced in ES6, which we will discuss below and are widely used today in our programming’s language.

\*\*\* There are two new ways of declaring variables that were introduced in ES6.

* **let**
* **const**

1. **Let:**

let is quite similar to var, in terms of usage. You can declare variables in exactly the same way, for example:

|  |
| --- |
|  |
|  | let myNewVariable = 2;  var myOldVariable = 3;  console.log(myNewVariable); // 2  console.log(myOldVariable); // 3   |  | | --- | | let is block scoped, while var is function scoped.  let cannot be accessed before declaration (var can be, and is actually the source of many bugs and confusion in the JS world).  let cannot be redeclared.   1. **Const:**   const is mostly the same as let (so has the same features, compared to var), but with one major difference. While let can be reassigned, constcannot — so it is a variable that always has one, and only one, value that must be assigned at the time of declaration, not later in any case. Let’s look at this example: | | const myConstVariable = 2;  let myLetVariable = 3;  console.log(myConstVariable); // 2  myLetVariable = 4; // ok  myConstVariable = 5; //wrong - TypeError thrown |  | |
|  | *Is Const === immutability? Has every one as doubt like it is primitive data type let see*  But there is one gotcha with const. While variables cannot be reassigned, it doesn’t make them truly immutable. If theconst variable has an array or object as its value, then you may change the content as much as you like:  const myConstObject = {mutableProperty: 2};  // myConstObject = {}; - TypeError  myConstObject.mutableProperty = 3; //ok  console.log(myConstObject.mutableProperty); // 3  const myConstArray = [1];  // myConstArray = []; - TypeError  myConstArray.push(2) //ok  console.log(myConstArray); // [1, 2] |

Note: This of course doesn’t apply to primitives like string, number, Boolean, etc., as they are immutable by nature.

*Immutability, here we come*

If you want our variables to be truly immutable, feel free to use tools like Object.freeze(), which would make the object immutable (or array — in fact array IS the object in JS). Unfortunately, it would only be shallow immutability, so if you have nested objects, they can still be modified:

/\* object.freeze work like below \*/

var myGlobalObject = {mutableProperty: 2};

Object.freeze(myGlobalObject);

function makeGlobalFreeze(){

var myLocalObject = myGlobalObject;

console.log(myLocalObject);

myLocalObject.mutableProperty ="Hello";

console.log(myLocalObject);

console.log(myGlobalObject);

}

makeGlobalFreeze();

console.log(myGlobalObject);

**Global and function scoped variables***:*

 The scope of the variable determines where your variables are available. Taken from a different perspective, one can say that scope defines which variables (or functions) you can use in the specific area of code. Scope can be global (so things declared within a global scope are accessible everywhere in your code) or local. Obviously, locally scoped variables are accessible only within some boundaries.

// global scope

var globalVariable = 10;

function functionWithVariable() {

// local scope

var localVariable = 5;

console.log (globalVariable); // 10

console.log (localVariable); // 5

}

FunctionWithVariable ();

//global scope again

console.log (global Variable); // 10

console.log (local Variable); // undefined

Here, the variable global Variable has global scope, so can be easily accessed within our function or in any other area of our code. But the variable local Variable that is defined within a function is only visible inside it.

So, whatever is created inside a function can be accessed there and within all nested functions (it might even be nested multiple times).

**Hoisting in JavaScript**:

https://developer.mozilla.org/en-US/docs/Glossary/Hoisting

Now, there are a few drawbacks related to having function scope. Before I get into details, let me introduce one more definition: hoisting. Again, simply speaking, this is the JavaScript mechanism that “moves” all variables (and functions) declarations to the top of our scope. Look at the following code:

function func() {

console.log(localVariable); // undefined

var localVariable = 5;

console.log(localVariable); // 5

}

func();

Below example explains whole thing about scopes and let and const:

function func() {

// function scope

let localVariable = 5;

var oldLocalVariable = 5;

if (true) {

// block scope

let nestedLocalVariable = 6;

var oldNestedLocalVariable = 6;

console.log(nestedLocalVariable); // 6

console.log(oldNestedLocalVariable); // 6

}

// those are stil valid

console.log(localVariable); // 5

console.log(oldLocalVariable); // 5

// and this one as well because the it is declared as var, which is function scope

console.log(oldNestedLocalVariable); // 6

// but this on isn't because the it is declared as let, which is blcok scope

console.log(nestedLocalVariable); // ReferenceError: nestedLocalVariable is not defined

}

Java script supports loose typing (no need to define the type of variable while declaring)

**Typeof operator is used to know the type the variable is of:**

Typeof(<value>);

Typeof(<variable>);

Example:

var value =42;

typeof(value) – output is number

var value =” hello”;

typeof(value) –output is string

var a =true;

typeof(a) –output is Boolean

This is bug is JavaScript

var a =null; output is

typeof(a);

"object"

**Example of type of operators:**

var test1 = 1;

var test2 = "Something";

var test3 = true;

var test4 = {};

var test5 = new Array();

var test6 = new Date();

var test7;

var test8 = null;

alert(typeof (test1)); //number

alert(typeof (test2)); //string

alert(typeof (test3)); //Boolean

alert(typeof (test4)); //object

alert(typeof (test5)); //object

alert(typeof (test6)); //object

alert(typeof (test7)); //undefined

alert(typeof (test8)); //object

**Type coercion and the == operator**

Concatenation with string values

123 + “4” = “1234”

One value is number and one is string, interpreter what it will do, it will convert on of the value to string and output is “1234”

The == operator, single = is usually takes as assignment operator

When you want to compare the variables we have use “==”

var a = 10;

var b = 10;

if (a == b) {

console.log (“both are equal”);}

output is both are equal;

var a =10;

var b ="10";

if(a==b) {console.log("equals")};

output is equals

The === operator check the type too

Var a = 10;

If(a) {

Cosole.log (“is true”)}

Output is true

**Java script Functions**:

* JavaScript, functions are first-class objects
* They can be referenced by variables
* Declared with literals
* Even passed as function parameters
* FUNCTIONS AS FIRST-CLASS OBJECTS
* Objects in JavaScript enjoy certain capabilities:
* They can be created via literals.
* They can be assigned to variables, array entries, and properties of other objects.
* They can be passed as arguments to functions.
* They can be returned as values from functions.
* They can possess properties that can be dynamically created and assigned.

**What is a Function?**

A function is a *subprogram* designed to perform a particular task.

Functions are executed when they are called. This is known as *invoking* a function.

Values can be *passed* into functions and used within the function.

Functions *always* return a value. In JavaScript, if no return value is specified, the function will return undefined.

**Functions are *objects*.**

**There are three type of functions in JavaScript**:

* **Named functions**
* **Anonymous functions**
* **Immediately invoked functions**

1. **Named Functions**:

function testExample(){

var test ="hey i am test function";

return test;

}

console.log(testExample());

Anonymous Functions:

var textExample = function(a){

var greeting ="Hi My name is " +a;

return greeting;

}

var a ="Dinail"

console.log(textExample(a));

Examples:

let name = function(parameters){  
 statements  
}

let name = (parameters) => { // ES6 way of calling functions =>  
 statements  
}

**2. Immediately called functions:**

(function testExample(){

var test ="hey i am test function";

console.log(test);

}())

Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.

Primitive parameters (such as a number) are passed to functions by value; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function.

If you pass an object (i.e. a non-primitive value, such as [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

var x, y;

var mycar = {make: 'Honda', model: 'Accord', year: 1998};

x = mycar.make; // x gets the value "Honda"

function myFunc(theObject) {

theObject.make = 'Toyota';

}

myFunc(mycar);

y = mycar.make; // y gets the value "Toyota"

// (the make property was changed by the function)

**Closure in JavaScript:**

<https://medium.freecodecamp.org/javascript-closures-simplified-d0d23fa06ba4> -- best example for Closures.

**What is a closure?**

* A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function’s variables — a scope chain.
* The closure has three scope chains:
* it has access to its own scope — variables defined between its curly brackets
* it has access to the outer function’s variables
* it has access to the global variables
* To the uninitiated, this definition might seem like just a whole lot of jargon!

**But what really is a closure**?

**A Simple closure**

Let’s look at a simple closure example in JavaScript:

function outer() {

var b = 10;

function inner() {

var a = 20;

console.log(a+b);

} return inner;

}

**Here we have two functions**:

an outer function outer which has a variable b, and returns the inner function

an inner function inner which has its variable called a, and accesses an outer variable b, within its function body

The scope of variable b is limited to the outer function, and the scope of variable a is limited to the inner function.

Let us now invoke the outer() function, and store the result of the outer() function in a variable X. Let us then invoke the outer() function a second time and store it in variable Y.

function outer() {

var b = 10;

function inner() {

var a = 20;

console.log(a+b);

}

return inner;

}

var X = outer(); //outer() invoked the first time

var Y = outer(); //outer() invoked the second time

Let’s see step-by-step what happens when the outer() function is first invoked:

Variable b is created, its scope is limited to the outer() function, and its value is set to 10.

The next line is a function declaration, so nothing to execute.

On the last line, return inner looks for a variable called inner, finds that this variable inner is actually a function, and so returns the entire body of the function inner.

[Note that the return statement does not execute the inner function — a function is executed only when followed by () — , but rather the return statement returns the entire body of the function.]

The contents returned by the return statement are stored in X.

Thus, X will store the following:

function inner() {

var a=20;

console.log(a+b);

}

Function outer() finishes execution, and all variables within the scope of outer() now no longer exist.

This last part is important to understand. Once a function completes its execution, any variables that were defined inside the function scope cease to exist.

The lifespan of a variable defined inside of a function is the lifespan of the function execution.

What this means is that in console.log(a+b), the variable b exists only during the execution of the outer() function. Once the outer function has finished execution, the variable b no longer exists.

When the function is executed the second time, the variables of the function are created again, and live only up until the function completes execution.

Thus, when outer() is invoked the second time:

A new variable b is created, its scope is limited to the outer() function, and its value is set to 10.

The next line is a function declaration, so nothing to execute.

return inner returns the entire body of the function inner.

The contents returned by the return statement are stored in Y.

Function outer() finishes execution, and all variables within the scope of outer() now no longer exist.

The important point here is that when the outer() function is invoked the second time, the variable b is created a new. Also, when the outer() function finishes execution the second time, this new variable b again ceases to exist.

This is the most important point to realize. The variables inside the functions only come into existence when the function is running, and cease to exist once the functions completes execution.

Now, let us return to our code example and look at X and Y. Since the outer() function on execution returns a function, the variables X and Y are functions.

This can be easily verified by adding the following to the JavaScript code:

console.log(typeof(X)); //X is of type function

console.log(typeof(Y)); //Y is of type function

Since the variables X and Y are functions, we can execute them. In JavaScript, a function can be executed by adding () after the function name, such as X() and Y().

function outer() {

var b = 10;

function inner() {

var a = 20;

console.log(a+b);

}

return inner;

}

var X = outer();

var Y = outer();

//end of outer() function executions

X(); // X() invoked the first time

X(); // X() invoked the second time

X(); // X() invoked the third time

Y(); // Y() invoked the first time

When we execute X() and Y(), we are essentially executing the inner function.

Let us examine step-by-step what happens when X() is executed the first time:

Variable a is created, and its value is set to 20.

JavaScript now tries to execute a + b. Here is where things get interesting. JavaScript knows that variable a exists since it just created it. However, variable b no longer exists. Since b is part of the outer function, b would only exist while the outer() function is in execution. Since the outer() function finished execution long before we invoked X(), any variables within the scope of the outer function cease to exist, and hence variable b no longer exists.

How does JavaScript handle this?

Closures

The inner function can access the variables of the enclosing function due to closures in JavaScript. In other words, the inner function preserves the scope chain of the enclosing function at the time the enclosing function was executed, and thus can access the enclosing function’s variables.

In our example, the inner function had preserved the value of b=10 when the outer() function was executed, and continued to preserve (closure) it.

It now refers to its scope chain and notices that it does have the value of variable b within its scope chain, since it had enclosed the value of b within a closure at the point when the outer function had executed.

Thus, JavaScript knows a=20 and b=10, and can calculate a+b.

You can verify this by adding the following line of code to the example above:

/\* example of clouse \*/

function outer(){

var b =10;

function inner () {

var a =20;

console.log("The sum of a+b is " + (a+b));

}

return inner;

}

outer();

var x = outer();

var y = outer();

console.log("the outer function once executed is " + outer());

x();

x();

x();

y();

Open the Inspect element in Google Chrome and go to the Console. You can expand the element to actually see the Closure element (shown in the third to last line below). Notice that the value of b=10 is preserved in the Closure even after the outer() function completes its execution.

Variable b=10 is preserved in the Closure, Closures in JavaScript

Let us now revisit the definition of closures that we saw at the beginning and see if it now makes more sense.

So the inner function has three scope chains:

access to its own scope — variable a

access to the outer function’s variables — variable b, which it enclosed

access to any global variables that may be defined

Closures in Action:

To drive home, the point of closures, let’s augment the example by adding three lines of code:

function outer() {

var b = 10;

var c = 100;

function inner() {

var a = 20;

console.log("a= " + a + " b= " + b);

a++;

b++;

}

return inner;

}

var X = outer(); // outer() invoked the first time

var Y = outer(); // outer() invoked the second time

//end of outer() function executions

X(); // X() invoked the first time

X(); // X() invoked the second time

X(); // X() invoked the third time

Y(); // Y() invoked the first time

When you run this code, you will see the following output in the console.log:

a=20 b=10

a=20 b=11

a=20 b=12

a=20 b=10

Let’s examine this code step-by-step to see what exactly is happening and to see closures in Action!

var X = outer(); // outer() invoked the first time

The function outer() is invoked the first time. The following steps take place:

Variable b is created, and is set to 10

Variable c is created, and set to 100

Let’s call this b(first\_time) and c(first\_time) for our own reference.

The inner function is returned and assigned to X

At this point, the variable b is enclosed within the inner function scope chain as a closure with b=10, since inner uses the variable b.

The outer function completes execution, and all its variables cease to exist. The variable c no longer exists, although the variable b exists as a closure within inner.

var Y= outer(); // outer() invoked the second time

Variable b is created anew and is set to 10

Variable c is created anew.

Note that even though outer() was executed once before variables b and c ceased to exist, once the function completed execution they are created as brand new variables.

Let us call these b(second\_time) and c(second\_time) for our own reference.

The inner function is returned and assigned to Y

At this point the variable b is enclosed within the inner function scope chain as a closure with b(second\_time)=10, since inner uses the variable b.

The outer function completes execution, and all its variables cease to exist.

The variable c(second\_time) no longer exists, although the variable b(second\_time) exists as closure within inner.

Now let’s see what happens when the following lines of code are executed:

X(); // X() invoked the first time

X(); // X() invoked the second time

X(); // X() invoked the third time

Y(); // Y() invoked the first time

When X() is invoked the first time,

variable a is created, and set to 20

the value of a=20, the value of b is from the closure value. b(first\_time), so b=10

variables a and b are incremented by 1

X() completes execution and all its inner variables — variable a — cease to exist.

However, b(first\_time) was preserved as the closure, so b(first\_time) continues to exist.

When X() is invoked the second time,

variable a is created a new, and set to 20

Any previous value of variable a no longer exists, since it ceased to exists when X() completed execution the first time.

the value of a=20

the value of b is taken from the closure value — b(first\_time)

Also note that we had incremented the value of b by 1 from the previous execution, so b=11

variables a and b are incremented by 1 again

X() completes execution and all its inner variables — variable a — cease to exist

However, b(first\_time) is preserved as the closure continues to exist.

When X() is invoked the third time,

variable a is created anew, and set to 20

Any previous value of variable a no longer exists, since it ceased to exist when X() completed execution the first time.

the value of a=20, the value of b is from the closure value — b(first\_time)

Also note that we had incremented the value of b by 1 in the previous execution, so b=12

variables a and b are incremented by 1 again

X() completes execution, and all its inner variables — variable a — cease to exist

However, b(first\_time) is preserved as the closure continues to exist

When Y() is invoked the first time,

variable a is created a new, and set to 20

the value of a=20, the value of b is from the closure value — b(second\_time), so b=10

variables a and b are incremented by 1

Y() completes execution, and all its inner variables — variable a — cease to exist

However, b(second\_time) was preserved as the closure, so b(second\_time) continues to exist.

**Objects in JavaScript:**

**Objects can be created by using 3 ways in JavaScript**:

* **Using object initializers**
* **Using a constructor function**
* **Using the Object.Create method**

1. **Using object initializers:**

Var myObj = {}; -- empty object with curly brasses

Console.log(myObj); --output object { }

Object is a collection of data and functionality

Data usually consists of properties

Functionality usually consist of methods or functions

myObj.prop="hello";

"hello"

console.log(myObj);

Output: {prop: "hello"}

myObj.prop2=123;

123

console.log(myObj);

Output: {prop: "hello", prop2: 123}

Console.log (“The number property is “+ myObj. Prop2);

Output: The number property is 123

var myobj = { "prop": "hello", "prop1": 123, "prop2":true};

console.log(myObj);

output :1 {prop: "hello", prop2: 123}

Object properties can be accessible directly

New properties can be adding on object directly

[JavaScript Objects and Prototypes In-depth](https://www.youtube.com/playlist?list=PLqq-6Pq4lTTaflXUL0v3TSm86nodn0c_u)

**Best example for objects**:

https://medium.com/@tkssharma/objects-in-javascript-object-assign-deep-copy-64106c9aefab

var myObj ={} – Empty object/inline object

var myObj= {

“name”: “joy”,

“age”:30;

“address”: {

“street”: “123 jio street”,

“city”: “US”

}

}

myObj.name – joy

myObj[“name”] – joy

when property name start with number then we use [] as a notation to access that property for the object

example:

var obj{

“name” :”pavan”,

“age” : “NA”,

“1” : “one”

}

Obj.1 // output error because here dot notation will not work

Obj[“1”]

The reason for this is that internally JavaScript actually has two different approaches for testing equality. Primitives like strings and numbers are compared by their value, while objects like arrays, dates, and plain objects are compared by their reference. That comparison by reference basically checks to see if the objects given refer to the same location in memory. Here is an example of how that works.

var x = {

firstName:"John",

lastName:"Doe",

age:50,

eyeColor:"blue"

};

var y = {

firstName:"John",

lastName:"Doe",

age:50,

eyeColor:"blue"

};

*// Outputs: false*

console.log(x==y);

*// Outputs: true*

console.log(x===y);

var a = JSON.stringify(x);

var b = JSON.stringify(y);

alert(a==b);

alert(a===b);

// Display some data from the object:

document.getElementById("demo").innerHTML =

x.age + " is " + x.age + " years old.";

alert(typeof(x));

alert(typeof(y));

alert(x==y);

alert(x===y);

alert(x.firstName===y.firstName);

var userOne ={

email: 'ryu@yahoo.com',

name: 'Ryu',

login() {

console.log(this.email, 'has logged in')

},

logout(){

console.log(this.email, 'has logged out')

}

};

userOne.name –output Ryu,

userOne.emial –output [**ryu@yahoo.com**](mailto:ryu@yahoo.com)

If you want to create a new property inside the userOne object

userOne.age =’20’,

we are using square bracket [] notation for the below examples

userOne[“name”]; --output is ryu , if you want read the value dynamically then it works like this

var prop = “name”;

userOne[prop] – output will be ryu, we can change the property name as below using []

userOne[‘name’] = “pavan”;

cosole.log(userOne) –output will be update with property name

prop =’emial’, now if your try to get the prop value from userOne, then the output will be email

userOne[prop] – output is “ryu.yahoo.com”

Class Methods:

if you want to create the different users and the data for the same, we need to copy the same code which creates lot of code, so instead we can use to create the users dynamically using constructors, which have same properties but different values

class User{

constructor(email, name){

this.email =email;

this.name =name;

}

login(){

console.log(this.email, 'just loged in');

}

logout(){

console.log(this.email, 'just loged out');

}

}

The 'new’ keyword

creates a new empty object {}

-sets the value of ‘this' to be the empty object

-calls the constructor method

// creating new user by new keyword, which invokes constructor method

userOne = new User("hello@yahoo.com" ,"pavan", "30");

userTwo = new User("example@yahoo.com" ,"example", "300");

console.log(userOne);

console.log(userTwo);

userOne.login();

userTwo.logout();

Method Chaining:

If you want to print the message in the console/ normal from the Methods or functions, we have to call like this

class User{

constructor(email, name){

this.email =email;

this.name =name;

}

login(){

console.log(this.email, 'just loged in');

}

logout(){

console.log(this.email, 'just loged out');

}

}

// creating new user by new keyword, which invokes constructor method

userOne = new User("hello@yahoo.com" ,"pavan", "30");

userTwo = new User("example@yahoo.com" ,"example", "300");

userOne.login()

userOne.logout()

which have multiple lines of code, instead we can call like

userOne.login().logout() – it give error

Instead you do something like this, just return the ‘this’ keyword form the login, logout, updateScore methods

class User{

constructor(email, name){

this.email =email;

this.name =name;

this.score = 0;

}

login(){

console.log(this.email, 'just loged in');

return this;

}

logout(){

console.log(this.email, 'just loged out');

return this;

}

updateScore(){

this.score ++;

console.log(this.email, ‘update score is’ this.score);

return this;

}

}

userOne.updateScore () – if we call this function no of time the score will increases

Note: Please the difference in code for class methods and method chaining, we are returning the ‘this’ keyword, in this case we can call the method in the same line which reduces code

userOne.login().updateScore().login() – output will be

hello@yahoo.com just loged in

class.js:19 hello@yahoo.com score is now 1

class.js:14 hello@yahoo.com just loged out

**Class Inheritance:**

Inheriting the class User to other user as below

class Admin extends User {

deleteUser(user){

users = users.filter( u => {

return u.email != user.email

})

}

}

Now Admin has access to all the prop ties and methods from the User class, but delete method is only access from the admin class not form the user class

// New admin user created as below

var admin = new Admin("example2@yahoo.com", "hello2");

//user array

var users=[ userOne, userTwo, admin];

// deleting user from the deleteUser Method

admin.deleteUser(userTwo);

console.log(users);

Note: admin user has been extended from the User class, no need to create the new constructor

constructor(email, name){

this.email =email;

this.name =name; }

we can create the new admin object by calling like this

var admin = new Admin("example2@yahoo.com", "hello2");

Consolelogs :

console.log(userOne);

console.log( userTwo);

console.log(admin);

var users=[ userOne, userTwo, admin];

admin.deleteUser(userTwo);

console.log(users)

Output:

User {email: "hello@yahoo.com", name: "pavan", score: 0}

User {email: "example@yahoo.com", name: "example", score: 0}

Admin {email: "example2@yahoo.com", name: "hello2", score: 0}

[User, Admin]

**Using a constructor function:**

**Constructors in JavaScript:**

In JavaScript actually there no class function, so we can call the create the constructors in the below ways. No longer we are using classes instead we can create the instances of User and call the user by new keyword below.

function User (email, name) {

this.email = email;

this.name = name;

this.login = function ()

{

console.log(this.email, 'is logged in ');

}

}

// userOne & userTwo are instances of User function

var userOne = new User("userOne@yahoo.com", "userOne");

var userTwo = new User("userTwo@yahoo.com", "userTwo");

userOne.login();

userTwo.login();

output from console:

*User {email: "userOne@yahoo.com", name: "userOne", login: ƒ}*

email:"userOne@yahoo.com"

login:*ƒ ()*

name:"userOne"

\_\_proto\_\_:Object

Note: Login method is created inside the object directly, previously if you remember, when we create the constructor using class, the methods (login & logout) used to create under \_\_Proto\_\_

Prototype in JavaScript:

Below code show how to create the methods to show under \_\_proto\_\_ which can we created as normal methods (like Sort, in linewidth, etc...)

function User (email, name) {

this.email = email;

this.name = name;

this.online = false;

}

User.prototype.login = function() {

this.online = true;

console.log(this.email, 'has logged in ');

}

User.prototype.logout = function() {

this.online = true;

console.log(this.email, 'has logged out');

}

var userOne = new User("userOne@yahoo.com", "userOne");

var userTwo = new User("userTwo@yahoo.com", "userTwo");

Now we can call those methods as usual like before

//console.log(userOne.login());

userTwo.login();

userTwo.logout();

userOne.login()

userOne;

output will be like below:

userOne

*User {email: "userOne@yahoo.com", name: "userOne", online: false}*

email:"userOne@yahoo.com"

name:"userOne"

online:false

\_\_proto\_\_:

login:*ƒ () // now login method is created under this*

constructor:*ƒ User(email, name)*

\_\_proto\_\_:Object

Prototype Inheritance:

Prototype inheritance is nothing but, how we are inheriting the class form another class. We can see the below code for the same.

/\* constructor function \*/

function User (email, name) {

this.email = email;

this.name = name;

this.online = false;

}

/\* proto type the methods as we see in the above topic Prototype in JavaScript to see the method under \_\_proto\_\_ when the object is created \*/

User.prototype.login = function() {

console.log(this.email, 'is logged in ');

}

User.prototype.logout = function() {

console.log(this.email, 'is logged out ');

}

/\* function syntax to inherit the prototype functions \*/

function Admin(...args){

//console.log(args) // args contains all the args form user (example email, name, online)

User.apply(this, args); // is used to apply all the properties form the user

this.role ="superadmin"; // If you want to create any new property for the new admin function we can create like this

}

/\* syntax to inherit the methods from the user function \*/

Admin.prototype = Object.create(User.prototype);

Admin.prototype.deleteUser = function(u){

users = users.filter( user => {

return user.email != u.email;

})

}

var userOne = new User("userOne@yahoo.com", "userOne");

var userTwo = new User("userTwo@yahoo.com", "userTwo");

var admin = new Admin("admin@yahoo.com", "admin")

var users =[userOne, userTwo, admin];

console.log(admin);

3.Using the Object. Create method

Objects can also be created using the [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create) method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

**// Animal properties and method encapsulation**  
**var Animal = {**  
 **type: 'Invertebrates', // Default value of properties**  
 **displayType: function() { // Method which will display type of Animal**  
 **console.log(this.type);**  
 **}**  
**};**

**// Create new animal type called animal1**   
**var animal1 = Object.create(Animal);**  
**animal1.displayType(); // Output:Invertebrates**

**// Create new animal type called Fishes**  
**var fish = Object.create(Animal);**  
**fish.type = 'Fishes';**  
**fish.displayType(); // Output:Fishes**

Copying of object in JavaScript:

The Native Way of Copying Objects

let obj = {  
 a: 1,  
 b: 2,  
};  
let copy = obj;  
  
obj.a = 5;  
console.log(copy.a);  
// Result   
// a = 5;  
// copy reference of object both object pointing to same copy

Using Object.assign() method

The **Object.assign**() method is used to copy the values of all enumerable own properties from one or more source objects to a target object

let obj = {  
 a: 1,  
 b: 2,  
};  
let objCopy = Object.assign({}, obj);  
console.log(objCopy);  
// Result - { a: 1, b: 2 }

Well, this does the job so far. We have made a copy of obj. Let's see if immutability exist:

let obj = {  
 a: 1,  
 b: 2,  
};  
let objCopy = Object.assign({}, obj);

console.log(objCopy); // result - { a: 1, b: 2 }  
objCopy.b = 80;  
console.log(objCopy); // result - { a: 1, b: 80 }  
console.log(obj); // result - { a: 1, b: 2 }

In the code above, we changed the value of the property 'b' in objCopyobject to 80 and when we log the modified objCopy object in the console, the changes only apply to objCopy. The last line of code checks that the objobject is still intact and hasn't change. This implies that we have successfully created a copy of the source object without any references to it.

\*\* *This method has a flaw that it only does a shallow copy. It means that nested properties are still going to be copied by reference. Be careful about it.*

**Note**: But Object.assing() will not work for nested object created, if we changed the nested properties in the copied object the changes will reflect in original object see below example:

let obj = {  
 a: 1,  
 b: {  
 c: 2,  
 },  
}  
let newObj = Object.assign({}, obj);  
console.log(newObj); // { a: 1, b: { c: 2} }

obj.a = 10;  
console.log(obj); // { a: 10, b: { c: 2} }  
console.log(newObj); // { a: 1, b: { c: 2} }

newObj.a = 20;  
console.log(obj); // { a: 10, b: { c: 2} }  
console.log(newObj); // { a: 20, b: { c: 2} }

newObj.b.c = 30;  
console.log(obj); // { a: 10, b: { c: 30} }  
console.log(newObj); // { a: 20, b: { c: 30} }

// Note: newObj.b.c = 30; Read why.. // this will be overcome by JSON Parse

Using JSON.parse(JSON.stringify(object));

This fixes the issue we had earlier. Now newObj.b has a copy and not a reference! This is a way to deep copy objects. Here's an example:

let obj = {   
 a: 1,  
 b: {   
 c: 2,  
 },  
}

let newObj = JSON.parse(JSON.stringify(obj));

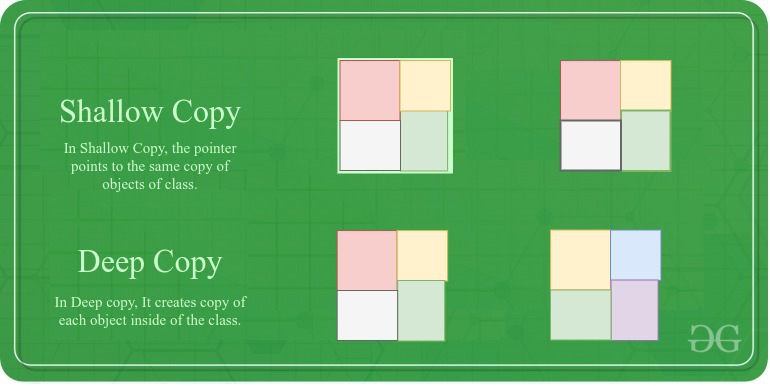
obj.b.c = 20;  
console.log(obj); // { a: 1, b: { c: 20 } }  
console.log(newObj); // { a: 1, b: { c: 2 } } (New Object Intact!)

JavaScript Concepts:

Shallow copy and Deep copy:

Shallow Copy: Shallow repetition is quicker. However, it’s “lazy” it handles pointers and references. Rather than creating a contemporary copy of the particular knowledge the pointer points to, it simply copies over the pointer price. So, each the first and therefore the copy can have pointers that reference constant underlying knowledge.

Deep Copy: Deep repetition truly clones the underlying data. It is not shared between the first and therefore the copy.



Below is the tabular Difference between the Shallow Copy and Deep Copy:

| Shallow Copy | Deep Copy |
| --- | --- |
| Shallow Copy stores the references of objects to the original memory address. | Deep copy stores copies of the object’s value. |
| Shallow Copy reflects changes made to the new/copied object in the original object. | Deep copy doesn’t reflect changes made to the new/copied object in the original object. |
| Shallow Copy stores the copy of the original object and points the references to the objects. | Deep copy stores the copy of the original object and recursively copies the objects as well. |
| Shallow copy is faster. | Deep copy is comparatively slower. |

Example:

Primitive data types

Primitive data types include the following:

Number — e.g. 1

String — e.g. 'Hello'

Boolean — e.g. true

undefined

null

When you create these values, they are tightly coupled with the variable they are assigned to. They only exist once. That means you do not really have to worry about copying primitive data types in JavaScript. When you make a copy, it will be a real copy. Let’s see an example:

const a = 5

let b = a // this is the copy

b = 6

console.log(b) // 6

console.log(a) // 5

By executing b = a , you make the copy. Now, when you reassign a new value to b, the value of b changes, but not of a.

Composite data types — Objects and Arrays

Now, if we make a copy b = a , and change some nested value in b, it actually changes a’s nested value as well, since a and b actually point to the same thing.

Example:

const a = {

en: 'Hello',

de: 'Hallo',

es: 'Hola',

pt: 'Olà'

}

let b = a

b.pt = 'Oi'

console.log(b.pt) // Oi

console.log(a.pt) // Oi

Objects

There are multiple ways to make copies of objects, especially with the new expanding and improving JavaScript specification.

Spread operator

Introduced with ES2015, this operator is just great, because it is so short and simple. It ‘spreads’ out all of the values into a new object. You can use it as follows:

const a = {

en: 'Bye',

de: 'Tschüss'

}

let b = {...a}

b.de = 'Ciao'

console.log(b.de) // Ciao

console.log(a.de) // Tschüss

Object.assign

This was mostly used before the spread operator was around, and it basically does the same thing. You have to be careful though, as the first argument in the Object.assign() method actually gets modified and returned. So make sure that you pass the object to copy at least as the second argument. Normally, you would just pass an empty object as the first argument to prevent modifying any existing data.

const a = {

en: 'Bye',

de: 'Tschüss'

}

let b = Object.assign({}, a)

b.de = 'Ciao'

console.log(b.de) // Ciao

console.log(a.de) // Tschüss

Pitfall: Nested Objects

As mentioned before, there is one big caveat when dealing with copying objects, which applies to both methods listed above. When you have a nested object (or array) and you copy it, nested objects inside that object will not be copied, since they are only pointers / references. Therefore, if you change the nested object, you will change it for both instances, meaning you would end up doing a shallow copy again. Example:// BAD EXAMPLE

const a = {

foods: {

dinner: 'Pasta'

}

}

let b = {...a}

b.foods.dinner = 'Soup' // changes for both objects

console.log(b.foods.dinner) // Soup

console.log(a.foods.dinner) // Soup

To make a deep copy of nested objects, you would have to consider that. One way to prevent that is manually copying all nested objects:

const a = {

foods: {

dinner: 'Pasta'

}

}

let b = {foods: {...a.foods}}

b.foods.dinner = 'Soup'

console.log(b.foods.dinner) // Soup

console.log(a.foods.dinner) // Pasta

In case you were wondering what to do when the object has more keys than only foods , you can use the full potential of the spread operator. When passing more properties after the ...spread , they overwrite the original values, for example const b = {...a, foods: {...a.foods}} .

Immediately called function and not :

(function(){

            console.log("Hello I am Immedialty called funciton");

        })();

        function sayhello() {

            console.log("Hello I am not Immedialty called funciton");

        }

const array = ["one", "two", "three", "four", "five"];

const age = 18 + ""; // converting number to string

age;

typeof(age);

Array Helper Methods:

https://dev.to/epicosity/javascript-array-helper-methods-40mp

forEach

map

filter

find

every

some

reduce

forEach()

// using a for loop

const numbers = [1, 2, 3, 4, 5];

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

console.log(numbers[i])

}

// Output

// 1

// 2

// 3

// 4

// 5

Let’s refactor this using a classic function declaration.

numbers.forEach(function(number){

console.log(number);

});

// 1

// 2

// 3

// 4

// 5

// using forEach with arrow

numbers.forEach( number => console.log(number));

Remember — the forEach callback is that it can also take in a second parameter, index. This will keep track of the current index of the number.

numbers.forEach((number, index) => console.log(`Number: ${number} is at index: ${index}`));

// Number: 1 is at index: 0

// Number: 2 is at index: 1

// Number: 3 is at index: 2

// Number: 4 is at index: 3

// Number: 5 is at index: 4

Apply, Call and Bind functions:

1. **Bind**:

Function is to over, this issues

function sample(name) {

    console.log("Inside the Function", name);

}

const userComponent = {

    counter: 0,

    name: "pavan",

    increment: function() {

        this.sample(); -- this give error if we call through *this*

    },

}

let result = sample.bind(userComponent, "Pavan Kumar Reddy"); -- this solve the issue.

result();

console.log(userComponent.counter);

console.log(userComponent.name);

Bind method creates a new function and sets the “this” keyword to the specified object.

We can pass second argument as optional as optional arguments with comma separated values.

Syntax: function.bind(thisArg, optionalArugment);

**2. Call:**

2. **Call** method won’t create new function, instead it will executes immediately by setting the “this” keyword to the specified object.

We can pass second argument as optional arguments with comma separated values.

Syntax : function.call(thisArg, optionalAruguments)

Example:

function Box(height, width) {

this.height = height;

this.width = width;

}

function Box1(radius, diameter) {

this.radius = radius;

this.diameter = diameter;

}

let boxCon = new Box(30, 40);

console.log(boxCon);

let one = new Box(100, 100);

let two = new Box(200, 100);

let three = new Box(300, 100);

let four = new Box(400, 100);

let five = new Box(500, 100);

function widget(height, width, color, radius, diameter) {

Box.call(this, height, width);

Box1.call(this, radius, diameter);

this.color = color;

}

let newwidget = new widget(100, 200, "red", 20.00, "30cm");

console.log(newwidget);

3.**Apply** :

Apply is same as call, only thing we need to pass the array of argument instead of single arry:

Example:

const obj = {

name: "Pavan"

}

function applyCallBind(name) {

console.log("Inside the applyCallBind function written by -->", this.name, name);

}

let bindResult = applyCallBind.bind(obj); // bind will creat new function and return

bindResult("Kumar");

applyCallBind.call(obj, "REDDY");

let arr = ["Sogalpalli", "Kumar", "Reddy"]

applyCallBind.apply(obj, arr);

function Box(height, width) {

    this.height = height;

    this.width = width;

}

function Box1(radius, diameter) {

    this.radius = radius;

    this.diameter = diameter;

}

let boxCon = new Box(30, 40);

console.log(boxCon);

let one = new Box(100, 100);

let two = new Box(200, 100);

let three = new Box(300, 100);

let four = new Box(400, 100);

let five = new Box(500, 100);

function widget(height, width, color, radius, diameter) {

    Box.call(this, height, width);

    Box1.call(this, radius, diameter);

    this.color = color;

}

let newwidget = new widget(100, 200, "red", 20.00, "30cm");

console.log(newwidget);

const a = {

    one: "one",

    two: "two",

    three: "three"

}

const he = {

    four: "4",

    five: "5",

    six: "6"

}

console.log("this is a Object ", a);

let b = {...a };

console.log("this is newly created objec from a", b);

let arr = [1, 2];

let arr1 = [3, 4];

console.log("This original array::", arr);

let uniqueValues = [...new Set(arr)];

console.log("Unquie set of arry is:::", uniqueValues)

let objectAssign = Object.assign(a);

let mergeArry = {...arr, ...arr1 };

console.log("Newy created another object using Object.assign is ::", objectAssign);

console.log("Mergeed arrys are ::", mergeArry);

console.log(arr);

console.log(arr1);

let arr3 = [1, 2, 3];

let arr2 = [4, 5];

console.log(arr3);

console.log(arr2);

arr2 = [...arr3, ...arr2];

arr3.push(56);

console.log(arr3);

console.log(arr2);

obje = {...a, ...he }

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

**Callbacks:**

Callbacks are the functions which are send as an argument to another function and called later at some point of time when need.

*I will call back later!"*

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

Example:

function one(value1, cb) {

    console.log("Inside function one, the value is", value1 + 10);

    cb(value1 + 10, false);

}

function two(value2, cb) {

    console.log("Inside function one, the value is", value2 + 10);

    cb(value2 + 10, false);

}

function three(value3, cb) {

    console.log("Inside function one, the value is", value3 + 10);

    cb(value3 + 10, false);

}

one(10, function(result1, error) {

    if (!error) {

        two(result1, function(result2, error) {

            if (!error) {

                three(result2, function(result3, error) {

                    if (!error) {

                        console.log(result3);

                    }

                })

            }

        })

    }

})

**Generator Functions in ES6:**

* Generators are a special class of functions that simplify the task of writing iterators.
* A generator is a function that produces a sequence of results instead of a single value, i.e you generate ​a series of values.

**Normal function:**

function normalFunc() {

console.log('I')

return;

console.log('cannot')

console.log('be');

return;

console.log('stopped.')

}

normalFunc()

**to stop the normal function, we need to return from middle but for generator, it can resume and call from that point.**

**Example:**

function \* generatorFunction() { // Line 1

console.log('This will be executed first.');

yield 'Hello, '; // Line 2

console.log('I will be printed after the pause');

yield 'World!';

}

const generatorObject = generatorFunction();

generatorObject.next().value

generatorObject.next().value

**Promise in Java Script**:

* What is a Promise?
* A promise is an object that may produce a single value sometime in the future: either a resolved value, or a reason that it’s not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.