**Java script is an Object Oriented language**

Java script frames are below:

1. Angular
2. React
3. 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—

1. No need to declare variable type
2. The same variable can be assigned values of different types
3. 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**
2. **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:**

1. **Number**
2. **String**
3. **Boolean**
4. **undefined**
5. **null**

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

**Non-primitive types:**

1. **Objects**
2. **Functions**
3. **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.

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

**Typeof operator** is used to know the which 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

**Functions in JavaScript**:

**What is a Function**?

1. A function is a *subprogram* designed to perform a particular task.
2. Functions are executed when they are called. This is known as *invoking* a function.
3. Values can be *passed* into functions and used within the function.
4. Functions *always* return a value. In JavaScript, if no return value is specified, the function will return undefined.
5. Functions are *objects*.

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

1. Named functions
2. Anonymous functions
3. Immediately invoked functions

**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  
}

**Immediately called functions**:

(function testExample(){

var test ="hey i am test function";

console.log(test);

}())

* 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)

**Objects in JavaScript:**

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}

1. Object properties can be accessible directly
2. New properties can be adding on object directly

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

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”]**

Event bubbling/Event capturing:

**https://javascript.info/bubbling-and-capturing**

Compare of objects:

Example:

**http://adripofjavascript.com/blog/drips/object-equality-in-javascript.html**

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]

**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:

1. *User {email: "userOne@yahoo.com", name: "userOne", login: ƒ}*
   1. email:"userOne@yahoo.com"
   2. login:*ƒ ()*
   3. name:"userOne"
   4. \_\_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}*

* 1. email:"userOne@yahoo.com"
  2. name:"userOne"
  3. online:false
  4. \_\_proto\_\_:
     1. login:*ƒ () // now login method is created under this*
     2. constructor:*ƒ User(email, name)*
     3. \_\_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, onine**)

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);