**ECMAScript 6 (2015)**

* ECMAScript is a language that JavaScript is made based on that. Features can be added to this language but that does not mean JavaScript will quickly be affected.
* JavaScript is one implementation of ECMAScript.
* We can do without the **function** keyword in today's JavaScript code. We can use arrow functions for functions that don't need to reference **this** and use the method shorthand (sayHello(){}) for anything that needs to reference **this**.
* A polyfill is **a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it**. The polyfill uses non-standard features in a certain browser to give JavaScript a standards-compliant way to access the feature.

**Let and const**

* let was introduced because of block scope problems

\* function func() {

var x = 10;

if(true) {

var x = "Derry";

console.log('Derry');

}

console.log(x)

}

func()// x is logged Derry which is wrong

* const // the type and value can't be changed but if it's an object or an array we can change the items.

const NAMES = { age : 23 };

NAMES.age = 20;// ok

NAMES = { } //not ok

NAMES.name = 'Derry' //ok

console.log(NAMES);

**Arrow functions**

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow\_functions

* It helps with the reduction of code. Remove the function keyword. and use =>

let func = ( x , y) => {

//code

return x\*y;

}

* If the code inside a function is just one line we can make it even shorter by removing braces and the return keyword

let func = ( x , y) => x\*y;

* If the return value is an object, we have to put it in parentheses

let func = ( x , y) => ({ x:x ,y: y });

* If the argument is just one value, we can remove the parentheses

array.map( item => item\*5);

* This keyword points to the current function. If we want to use the properties of a constructor function in an anonymous function inside that constructor function, we can use a global variable and put this keyword inside it, then use that variable. In ecam6 we can use arrow functions for this purpose.

function Person() {

this.age = 0;

that = this;

setInterval(function(){

that.age++;

console.log(that)

}, 1000);

}

let person = new Person()

function Person() {

this.age = 0;

setInterval(() => {

this.age++;

console.log(this)

}, 1000);

}

let person = new Person()

* In normal objects, {}, the behavior of **this** mentioned above is reversed.

let obj = {

a : 10,

b : function() {

console.log(this.a , this)

},

c : () => {

console.log(this.a , this)

}}

obj.b()//shows a = 10, this points to obj

obj.c()//a is undefined, the second output is window, the global variable, which means it points to the c function itself

* arguments is an object global variable, that can be accessed inside every function that is declared by function keyword. Argument[1] shows the first argument of that function. It's not accessible inside an arrow function.
* Arrow functions can't be used as constructor functions and they don't have any prototypes.

**Default parameters**

**Before ecma6**

function sum ( a , b ) {

x = a || 1

y = typeof b == 'undefined' ? 1 : b

return ( x + y )

}

**Ecma6**

function sum ( a = 1 , b = 1 ) {

return ( x + y )

}

**Rest operator … items to list**

* When we don't know how many arguments we may have, in normal functions we could use arguments to access arguments. But what about arrow functions?
* … c says the rest of the arguments, no matter how many, can be accessed in a list called c. It converts them to a list.

function func ( … args) {

console.log(args) // [] all as a list

}

function func ( a = 1 , b = 1 , … c) {

console.log( a, b ,c ) // a , b , []

}

* Rest should be at the end, we can't have another item after it

**Spread … list to items**

* Before ecma6 apply was used to pass the items of a list as separate arguments.

function sum (a, b, c ) {

console.log(a + b + c)

}

var list = [1, 3, 5]

sum.apply(null, list)

* Spread converts the list to separate items

function sum (a, b, c ) {

console.log(a + b + c)

}

let list = [1, 3, 5]

sum(… list)

* It can be used to merge lists

let list1 = [1, 3, 5]

let list2 = [0, 7, 5]

let list3 = [… list1, … list2]

**For-of**

* It can be used to iterate through iterables. Like: lists, strings, map, set, generators. The object is not iterable.

let list = [1, 3, 5]

**for**(**let** item **of** list){

console.log(item)

}

**Destructuring**

* Is used in arrays and objects
* To easily put items of arrays and objects into separate variables

**Arrays Destructuring [ ]**

let list = [1, 3, 5]

let [item1, item2] = list // the first two items

* to put values in multiple variables in one line

let a , b

[a , b] = [1 , 2]

* Default values

let [a = 1, b = 1] = [3]

let a = 1, b = 1 // You can't put brackets here

[a , b] = [15 , 7]

console.log(a , b)

* It can be used to separate items of a list returned from a function
* Ignore some items

let list = [1, 3, 5]

let [item1, , item2] = list // The second item is ignored, the value of item2 will be 5

* we can use rest operator to put the rest of the items in a list

let list = [1, 3, 5, 9, 3, 5]

let [item1, , item2, …c] = list // The second item is ignored and c is [9, 3, 5]

**Object Destructuring { }**

* If the name of variables are identical to those of objects there is no need to write them in an orderly manner

let obj = {

name : "Derry",

}

let { age = 20 , foo : bar = () => { return 'Rose'} , name } = obj; // default values, and foo is given a new name with **:**

console.log(name , age , bar());

**[Template literals (Template strings) ` `](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)**

* Putting variables in strings and attaching them together was very troublesome before ecma6. We had to use + and that would be messy in long strings with too many variables.
* `` // this is backticks and it is called template literals. We can write in any number of lines with breaks which were not possible with a double quote. We can add `Hello` + var with a variable but there is a better solution:
* **`** string **${**var or mathematic calcualtions**}** string **`**
* It can be used with short-if

let obj = {

name : 'Derry'

}

console.log(`Hello ${obj.name}

Goodby`)

**Class**

* Classes are used to create and manage new objects and support inheritance—a key ingredient in object-oriented programming and a mechanism of reusing code.
* Classes in JavaScript are actually made by constructor functions, they just have a different look.
* Constructor() // is a reserved method for defining properties
* function keyword is not used to declare class methods.

**class** Car {

**constructor**(model, year){

this.model = model

this.year = year

}

run(){

console.log(this.model);

}

}

let car1 = new Car( "Benz");

car1.run();

**Inheritance In Class**

* extends keyword
* super #refers to the parent class. It can be used to access properties and methods in the parent class. We can also use it when we rewrite a property but we still may need the parent one.

**class** BMW **extends** Car {

**constructor**(model, year, color){

**super**(model, year)//It's like we say Car()

this.color = color

}

fly(){

console.log("fly");

}

run(){

**super**.run()

}

}

let car2 = new BMW ( "Benz", 1989, "blue");

car2.run();

* **This** keyword in inheritance: With **this** keyword, we also have access to the methods and properties of the parent class.

**class** BMW **extends** Car {

**constructor**(model, year, color){

**this.**model **=** model

this.color = color

}

fly(){

**this**.run();

}}

let car2 = new BMW ( "Benz", 1989, "blue");

car2.run();

**Static methods**

* without creating a new object of a class to access a method of a class, declare those methods as static methods. Static methods **can not be** accessed through objects of that class.

class BMW extends Car {

constructor(model, year, color){

super(model, year)

this.color = color

}

**static** run(){

console.log(**this**)

}}

**BMW.run()** //now it's possible. here **this.** is not an object.

let car1 = new BMW ( "Benz", 1989, "blue");

**car1.run();**// It executes the run in **parent** class, Car

* Inside a class, a method can not access static methods. Static methods can access static methods.

**Modifying Internal(native) classes**

class AdvanceArray extends **Array** {

find(value) {

let val = this.filter(item => item === value)

return val.length === 0 ? 'error' : val[0]

}}

let list = new AdvanceArray(derry , 'rosa' , 'sarah')

console.log(list.find("derry"))

console.log(list.reverse())

**Getter and Setter**

* In object-oriented languages setter and getter are used for encapsulation and for the purpose of indirect access to properties. Also, it can be used to make a property accessible only in certain places. For example only inside the class to prevent objects and children of that class access it. Or make it accessible inside the class and the children of that class, but objects of that class and objects of children can't access it. And so on.
* But in JavaScript, it is only used for a sense of indirect access to a property, because the other purposes are not implemented yet. Still, that property can be accessed directly.
* The most common use cases are (1) securing access to data properties and (2) adding extra logic to properties before getting or setting their values.
* set keyword // to create a method as a setter. An **object** can access this method like a **property** to **set** a value of a normal property, and it is only accessed in this way.
* get keyword // to create a method as a getter. An **object** can access this method like **property** to **get** a value inside a certain normal property, and it is only accessed in this way.

class Foo {

constructor (){

this.list = []

}

set setList (value){

this.list.push(value)

}

get getList (){

return (this.list.length === 0 ? undefined : this.list[this.list.length -1])

}

baz (){

return(`Inside baz ${this.getList}`) // access it in class

}

}

let bar = new Foo()

console.log(bar.list) //output: []

console.log(bar.getList) // output: undefined

bar.setList = 1

console.log(bar.list) // We still can access it, output: [1]

console.log(bar.getList) // output: 1

console.log(bar.baz()) // This also works, output: Inside baz 1

**Objects**

**Added features:**

let name = "Derry";

let keyname1 = "fullname 1"

let obj = {

**name,** // instead of name: name. When the name of the keyname is the same as the variable holding the value

**[**keyname1**]** : "Derry Johnson", // to use variable values as keyname use brackets

**"say hello"()** { // we can declare a method in objects this way, a string as the name of the method

return 'Hello Derry;

}

**sayGoodby()**{// also like methods in classes

}}

console.log(obj**[**'say hello'**]**); // To access a keyname having spaces and in string format we have to use brackets

console.log(obj**[**keyname1**]**);

**Added methods:**

* Object.assign(target object(base), second obj) // It is a static method added to the Object constructor function It merges two objects. It returns a new object and also alters the target object. In shared properties object 2 has priority.

let obj1 = { name: "Derry", family: "Johnson"}

let obj2 = { name: "Rose", family: "Pitt", age: "23"}

let obj3 = Object.aaign(obj1, obj2) // returns and alters obj2 into { name: "Rose", family: "Pitt", age: "23"}

* In Objects made out of classes, works the same and the prototype of the returned object is the class of target object
* Object.assign({}, any number of objects) // target is a normal empty object. It is used to prevent the change in our base objects. The prototype of returned {} is the base object and not any of the given objects. The priority of shared properties is from right to left.

class Class1{

constructor(){

this.name = "Derry"

}}

class Class2{

constructor(){

this.name = "Rose"

this.family = "Johnson"

}}

class Class3{

constructor(){

this.name = "Sarah"

this.age = 36

}}

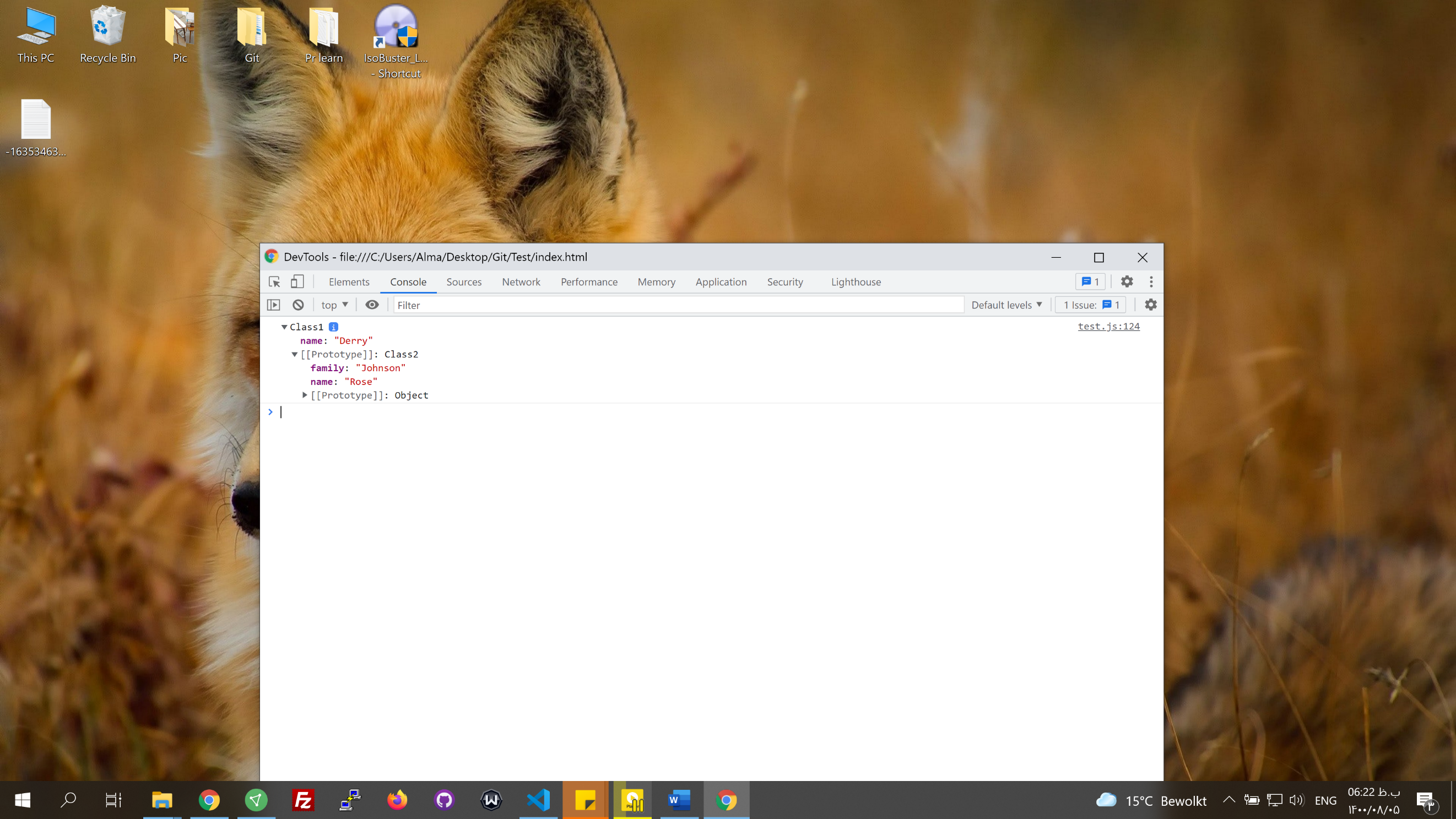
let obj1 = new Class1();

let obj2 = new Class2();

let obj3 = new Class3();

console.log(Object.assign({}, obj1, obj2, obj3))// output: { name: 'Sarah', family: 'Johnson', age: 36 }

* Object.setPrototypeOf(obj1, obj2) //Change the prototype of obj1 to obj2 and inherit properties of obj2



**Symbols**

* It's a data type
* They create unique values, that we don't know what it actually is, but it's always a new unique number
* To create a symbol we don't need the **new** keyword like what we do with string constructor functions.
* They can be used as identifiers for objects' properties.

let symbol1 = Symbol(''info'') //description is used to know what this symbol is made for

let prop1 = “name”

let obj = {

[prop1]: 'foo',

[symbol1]: 44

}

Obj.name // returns foo

// obj.Symbol(''info'') // We can't do this

obj.[symbol1]

* It has these characteristics
* We cannot access that unique number directly
* They are hidden to loops like for-in.
* To access them we should use the **variable name** that holds the symbol value inside it: obj.[symbol1]
* When is it used? For example when we want a unique key name for a property to hide it in the loops and prevent access to it by objects without the symbol.

**Shared Symbols**

* To access a symbol anywhere through its description
* We know the symbol Iis shared
* Symbol.for('description') // creates the symbol and under that descriptions stores it somewhere. We use this also to access it anywhere

**Well-known Symbols**

* They are pre-defined symbols that are built-in in JavaScript which is like ids that are used to change the behavior of some built-in methods.
* Symbol.replace // with this symbol we can change replace behavior completely.

class ReplaceX {

constructor(value) {

this.value = value;

}

[Symbol.replace](string) {

return `/${string}/${this.value}`;// It returns this as replace method returned value

}

}

//console.log('foo1'.repalce('foo1', 'bar')// replaces foo with bar

console.log('foo1'.replace(new ReplaceX('bar')))

**New methods in arrays**

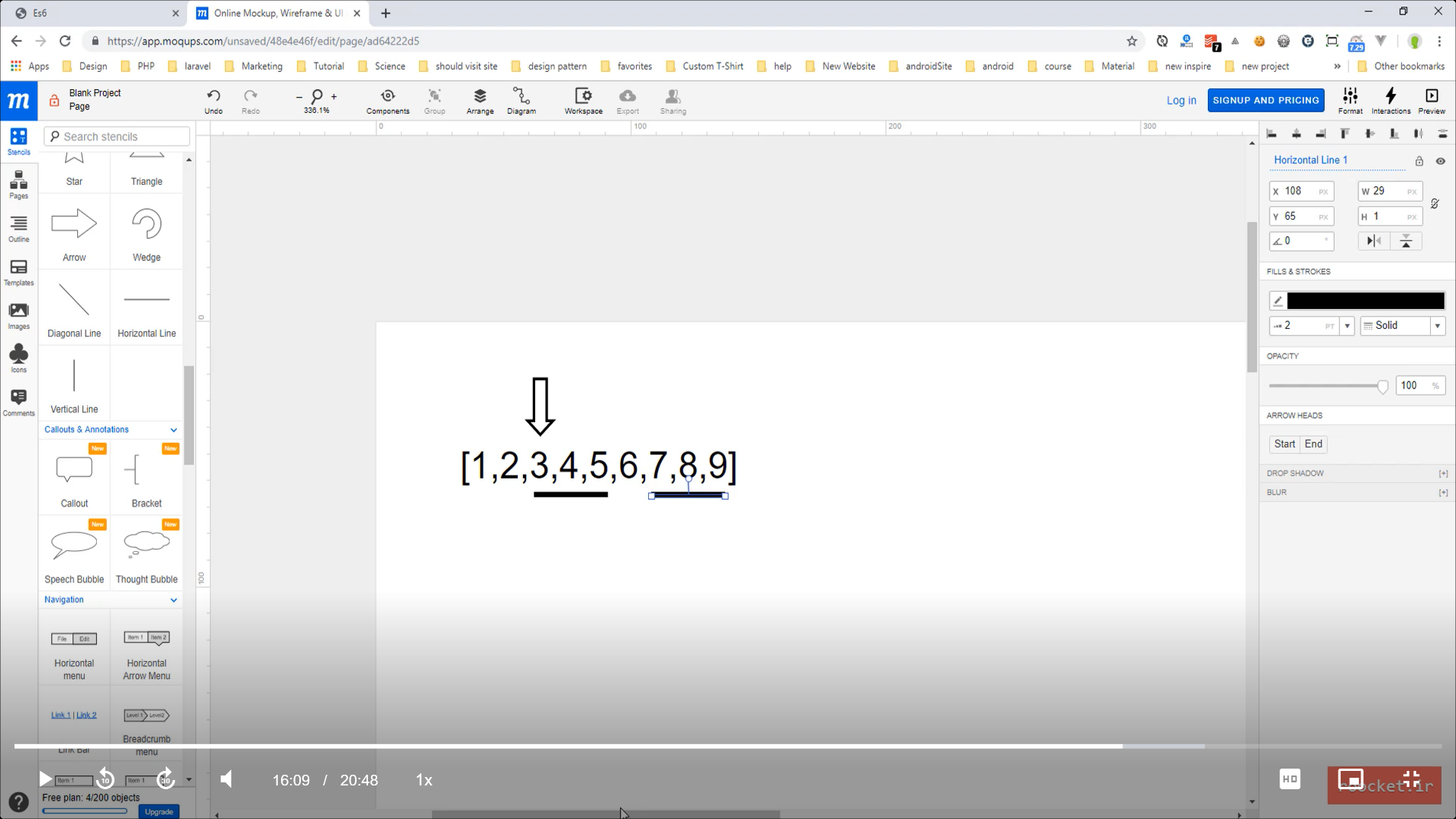
* Array.for(items) // If we use Array(2), there is a problem that it returns an empty list instead of a list holding 2. It actually returns a list of undefined items and the length is the number given. This happens only with numbers and only one number. This problem is solved by **for** constructor function. For is actually a static method.
* Array.from(list, map(item => item\*2)) // It elevates the Readability
* list.find(item => item % 2 ==0) // the difference with filter is that, filter returns all the items that match the condition, while find returns the first one that matches and exists
* list.entries() // returns an iterator holding a group of separate lists of every item and its index. It is used to traverse the lists without the use of loops like for-of.

let list = [5,9,3]

list.enteries() // output => iterator of [0,5], [1,9], [2,3]

list.next() // it returns the next item every time it is called when reaches the end it returns done

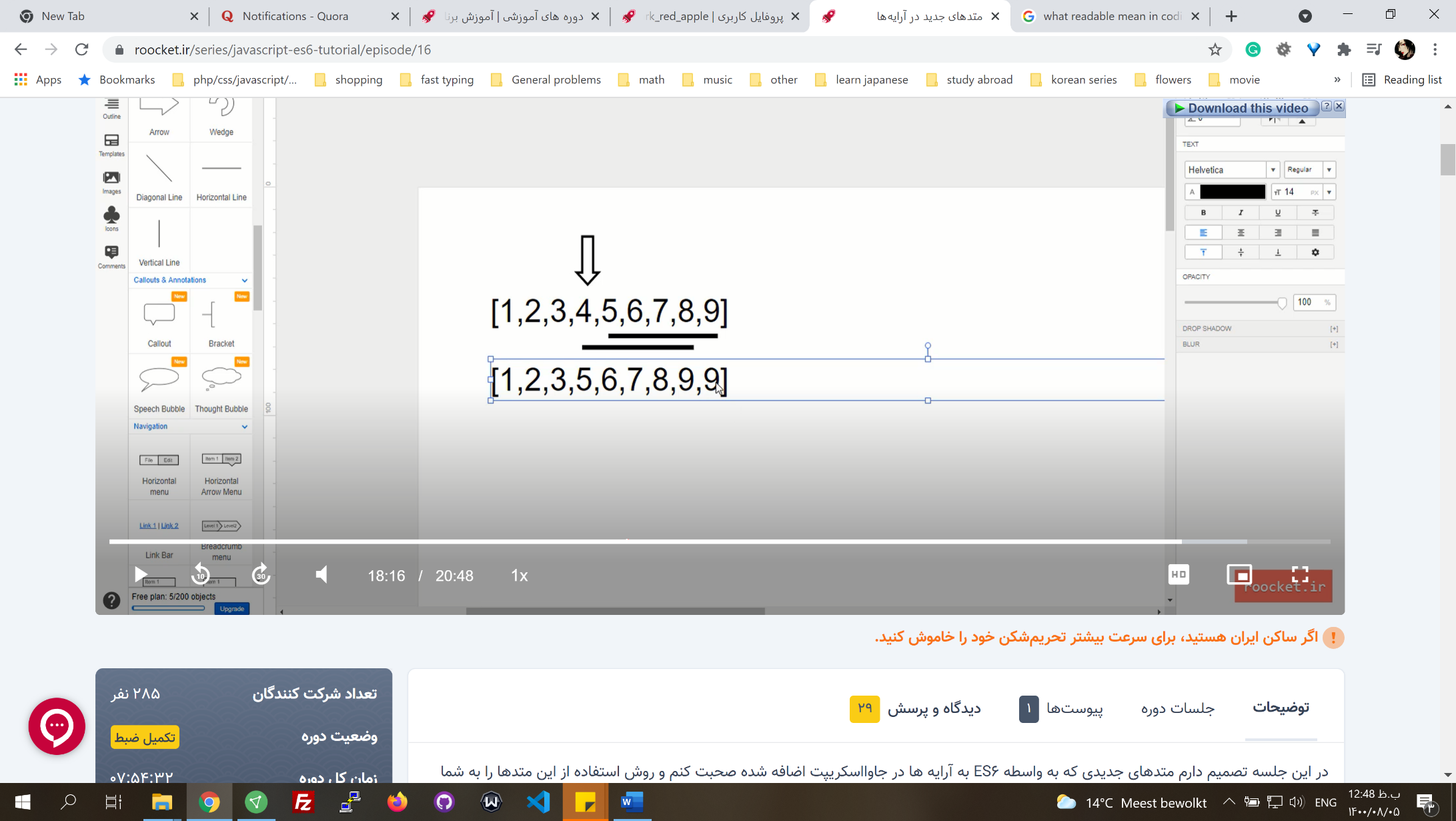
* list.fill(item,startindex,endindex) // it fills(replaces) all the items of **list** with the new item. Also, we can say which indexes we want it to fill. The item with endindex is not affected.
* List.copyWithin(targeindex,copystartindex,copyendinex) // is used in big arrays(buffers) to replace values of items with existing values inside the list. If we don't give the end index it will copy to the last index. The item with endindex is not included.



Copy this

To this

Target



**New methods in numbers**

* IsNaN // It's a global function. It checks if a number is NaN like (string + number). It also is found in Number.isNaN constructor function as a static method.
* Number.isFinite(num) // Checks if a number is finite.

let num = 1/0;

console.log(Number.isFinite(num)) // returns false

console.log(num) //returns infinity

* Number.isInteger(num)
* Math.sign(num) // Returnes 0, 1, -1, NaN if num is 0, positive, negative, NaN respectfully.
* Math.trunch() // It is similar to floor. Only difference is:

**New methods in strings**

* Str.includes(string, position) // searches a string in str. It is case-sensitive. The second parameter is optional which is the start index to look ahead. Return true or false.
* str.startswith(string, position) // It searches from start or given position, not in the whole string.
* str.endswith(string, position)

**Iterators**

* In JavaScript, an iterator is an object which defines a sequence and potentially a return value upon its termination. Specifically, an iterator is any object which implements the Iterator protocol by having a next() method that returns an object with two properties: value. The next value in the iteration sequence.
* Symbol.iterator // All the iterables have this method inside them. Which is a well-known function accessed by a symbol. We could access symbols by [ ].
* This method is used in for-of loops. It calls this method behind the scene and runs the next() function inside it, and gives the value to the variable we have defined. We can redefine this method by list[Symbol.iterator] = function() {}

let list = ["Derry", "Rose"]

console.log(list[Symbol.iterator]) //returns a function called values. It's native codes.

console.log(list[Symbol.iterator]())

for(let item of list) {

consol.log(item)

}

* Using this method, we can make objects iterable that by default are not.

let user = {

id : 1 ,

name : 'Derry' ,

email : 'Derry@gmail.com',

posts : **[** { id : 1, title : 'this is post one' }, { id : 2, title : 'this is post two'} **]**

}

**user[Symbol.iterator]** = function() {

// **this** here is the object that now holds user[Symbol.iterator] method too.

//**this**: {id: 1, name: 'Derry' , email: 'Derry @gmail.com', posts: Array(2), Symbol(Symbol.iterator): ƒ}

let posts = **this**.posts;

let step = 0;

return **{** // an object should be returned

**next**() { //next is the method that for runs, we have to define it

let **obj** = {

done : step >= posts.length, //done and value are musts

value : posts[step]

}

step++;

return **obj**;

} **}**}

for(let post of user) {

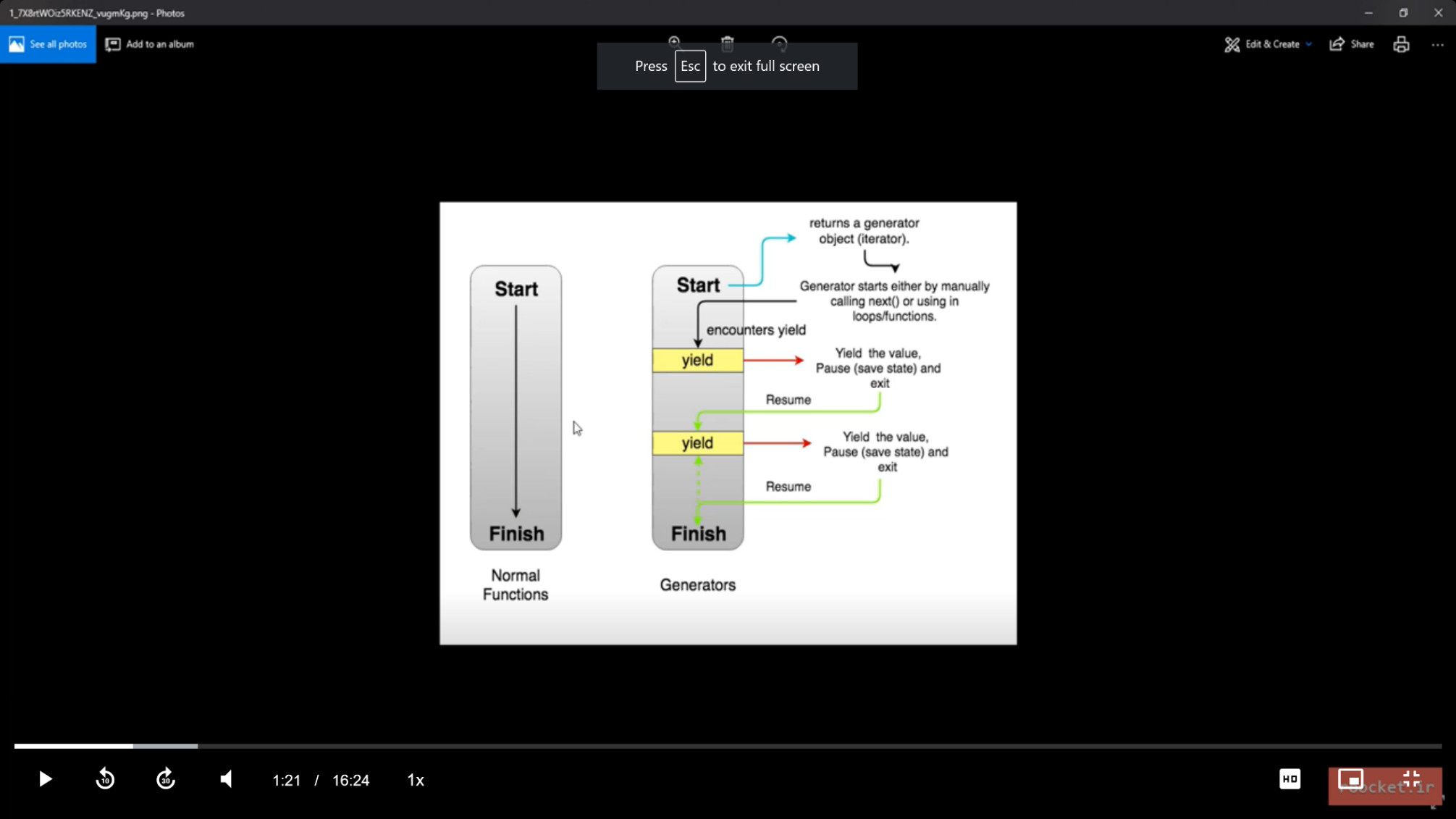
console.log(post);

}

* Iterators are used for the customized traverse. The performance of for-of and for with indexes in arrays is not that different in general. The primary purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container.

**Generators**

* Generators are a kind of function that has the ability to stop and continue at certain points. These points are marked by the keyword **yield**.
* It can be used when we need a value of that function at some point of our program and we need the next one at another forward point.
* When a generator has been defined an iterator is also created for it. This iterator is what is returned from this function and is actually what enables the stop and continue procedure. It calls next() that works based on yield points to run a part of code. This yield points can return a value.



* To declare a generator if :
  + It is a normal function put a **\*** after function keyword: function\*
  + It is a symbol method, put the **\*** before the opening bracket: \*[symbol.iterator]

function\* names(){

//code

yield 'Rose'

//code

yield 'Sarah'

//code

yield 'Derry'

return 'end' // when shows this the done value is true.

}

let namesIt = names() // It returns a an iterator

console.log(namesIt.next())

for (**const** name of names**()**){ // we have to loop through the returned iterator

console.log(name)

}

consol.log(…names()) // use the rest opertaor to put yield values in a list

* We can use them in objects instead of iterators to write cleaner and shorter codes.
* The iterator example with generators:

let user = {

id : 1 ,

name : 'Derry' ,

email : 'Derry@gmail.com',

posts : [ { id : 1, title : 'this is post one' }, { id : 2, title : 'this is post two'} ],

**\***[Symbol.iterator]() {

yield**\*** this.posts; // instead of two yields for the two items in posts we can use this

}

}}

for(let post of user) {

console.log(post);

}

**Callback Hell in ecma5**

http://callbackhell.com/

* The codes in JavaScript are interpreted from above forward. If an action needs time to be done, like connection to a server, or to a database, it is considered to be asynchronous. Asynchronous jobs run in the background while the rest of the code proceeds. Sometimes we need a value to be returned from these jobs. But return in usual approach doesn't work properly.

function doSomething() {

setTimeout(function(){ // a simulation for the job that is time consuming

**return 'code 2'**

},2000)

}

console.log('code 1')

console.log(**doSomething()**)// it does not show code 2

console.log('code 3')

* Callbacks handle these returned values. We define a function (called callback) as a **parameter** to pass to the original function, then it is called when is needed inside the body of that function, while we can pass data to it.

function doSomething(**callback**) {

setTimeout(function(){

**callback( 'code 2')**

},2000)

}

console.log('code 1')

doSomething(**funciton(data){**

**console.log(data)**

**}**)

console.log('code 3')

* The problem is that since callbacks are executed asynchronous, they become a hell of a nuisance to handle, considering errors, if we have too many of them entangled.

function doSomething(callback , error) {

setTimeout(() => {

let title = 'Apple';

if(!title) { error('err') return;}

setTimeout(() => {

let data = { user : ''}

if(!data) { error('err data'); return;}

callback(data);

}, 2000);

}, 2000);

}

console.log('run 1')

doSomething(function(data) {

console.log(data);

}, function(error) {

console.log(error);

})

console.log('run 3')

* How to handle them in ecma5: Keep the code shallow, give functions names and don't declare them as a parameter, and use function definition instead of assignments.

func1(data1, function(data2) {

data2.func2(function(){

//do something else

return data3 //do something

})

})

function func1(data1, callback){

//do something

callback(data2)

}

function func2(callback){

//do something

let outp = callback(); //returns data3

console.log(outp)

}

declare anonymous functions with function definition:

func1(data1, func1Callback)

function func1Callback(data2){

//do something

data2.func2(func2Callback)

}

function func2Callback(){

//do something

return data3

}

**Promise**

* Promises are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.
* They were introduced to remedy the problem with the chain of callbacks. It promises that retunes the data correctly and handles the error properly.
* Instead of declaring confusing chain of callbacks and handling their errors separately, chain of promises are used to improve readability and to simplify the way all the returned data and errors in this chain is handled.
* new Promise((resolve, reject) => { resolve(data); reject(error) }
* func.then(data => { }, err => { }).then().then() // multiple then to handle data for different promises
* func.catch(err => { }) // Bring the error out of then. one catch for all the promises
* finally

function func() {

return new **Promise**((resolve, reject) => {

//code

**reject**(error)

//code

**resolve**(data) //Is like we call the calleback

})

}

function func2() {

return new **Promise**((resolve, reject) => {

//code

**reject**(error)

//code

**resolve**(data)

})

}

func()

.**then**(data => {func2(data).then(data2 => {

console.log(data2)

})

})

.**catch**(error => console.log(error))

* Chain of promises //if one of the then methods returns another promise we can use another then to work on the data returned from the second promise

func() //func returns a promise

**.then**(data => func2(data))//then is a method in func that returns a promise by func2

**.then**(data2 => console.log(data2) //we handle the second promise returned values by the second then

**.catch**(error => console.log(error))

**Methods in Promise**

* Static resolve and reject methods

function doSomething(){

let name = "Derry"

if(err){

return **Promise.reject("An error happend")**

}

return **Promise.resolve(name)**

}

doSomething().then(data => console.log(data), err => consol.log(err) )

* Promise.all([]) // To perfume promises together and return their data in a list. It accepts an array of promises or non-promises. It performs all the promises. If a promise is rejected, no matter if all other promises are resolved, it always goes in rejected mode. If promises take different amounts of time to be completed, it waits for all of them to be completed. If all the promises are resolved, it returns an array of returned values, respectfully to their promises in the array given.
* Promise.race([ ]) //whatever that is done faster, rejected or resolved, it will be returned. If there is a nonasynchronous job like num = 432, it will return that.

let promise1 = new Promise((resolve , reject) => {

setTimeout(() => {

reject('rejected !!');

}, 2000);

})

let num = 432

let promise3 = new Promise((resolve , reject) => {

setTimeout(() => {

resolve('resolved !!');

}, 1000);

})

Promise.all([promise1, num, promise3])

.then(data => console.log(`all ${data}`))

.catch(err => console.log(`all ${err}`)) // it returns all rejected !!

Promise.race([promise1, promise3])

.then(data => console.log(`race ${data}`))

.catch(err => console.log(`race ${err}`)) // it returns race resolved !!

**Map**

* A collection data structure that enables key-value format.
* let map = new Map()
* map.set(key,value) //key can be string, number, symbol, function
* map.get(key)

let symbol = Symbol("My symbol")

let func = function (){}

let obj = {}

let myMap = new Map()

myMap.set('key1', 'Derry')

myMap.set(obj, 15)

myMap.set(func, 'rose')

myMap.set(83, null)

myMap.set(symbol, 'sarah')

myMap.get(obj)

* map.size // the number of items
* map.delet(item key)
* map.clear() // Deletes all data
* map.forEach((value, key) => { })
* map.has(key) //checks if that key exists in the map collection
* For-of

for (const [key, value] of map) {

consol.log(key,value)

}

* map.keys()
* map.value()
* map.entries() //It returns an iterator that we can loop through with next()
* Convert lists to map, a list should be in this format so we can convert it

let list = [['key1', 'Derry'], ['key2', 'Rose'], ['key3', 'Sarah']]

let map = new Map(list)

**Set**

* To create a collection of unique values.
* We can use it to create a set (of unique values) of an existing array
* let mySet = new Set([item1, item2, item3, item1 ])
* mySet.add(value)
* mySet.clear() // Clears all the items.
* mySet.delete(item) // Deletes that item
* mySet.entries() // Returns an iterator but the pair index-value is not the same as lists, for example. Here the index and value are the same, indexes are not numbers starting from 0, like how we had in arrays.
* mySet.keys()
* mySet.values() // the same as keys()
* mySet.has(item)
* … mySet // Separate the items
* [… mySet] // Convert to a list

**WeakMaps, WeakSets**

**WeakMaps**

* WeakMaps accept only **objects** {} as keys

let list = new WeakMap()

list.set({'name' : 'derry'}, 34)

* It's **not iterable**, so we can't use them in loops, they have fewer methods compared to maps.
* They are used to optimize **memory** usage. Map keeps the keys that are deleted later.

class Snow{

constructor(){

this.snowflake = new Array(100000).join('---')

}

}

window.snow = new Snow()

let myList = new WeakMap() //use WeakMap to remedy the problem

myList.set(window.snow, 'fall')

delete window.snow

console.log(window.snow)// This does not exist

console.log(myList)// But snow obj still exist as the key in myList, problem!

**WeakSats**

* It only accepts objects as items

let myList = new WeakSet(**[**{'item1': 'Derry'}, {'item2}: 'Sarah'}**]**)

* Like weakMap it deletes the item if the item source object is deleted.

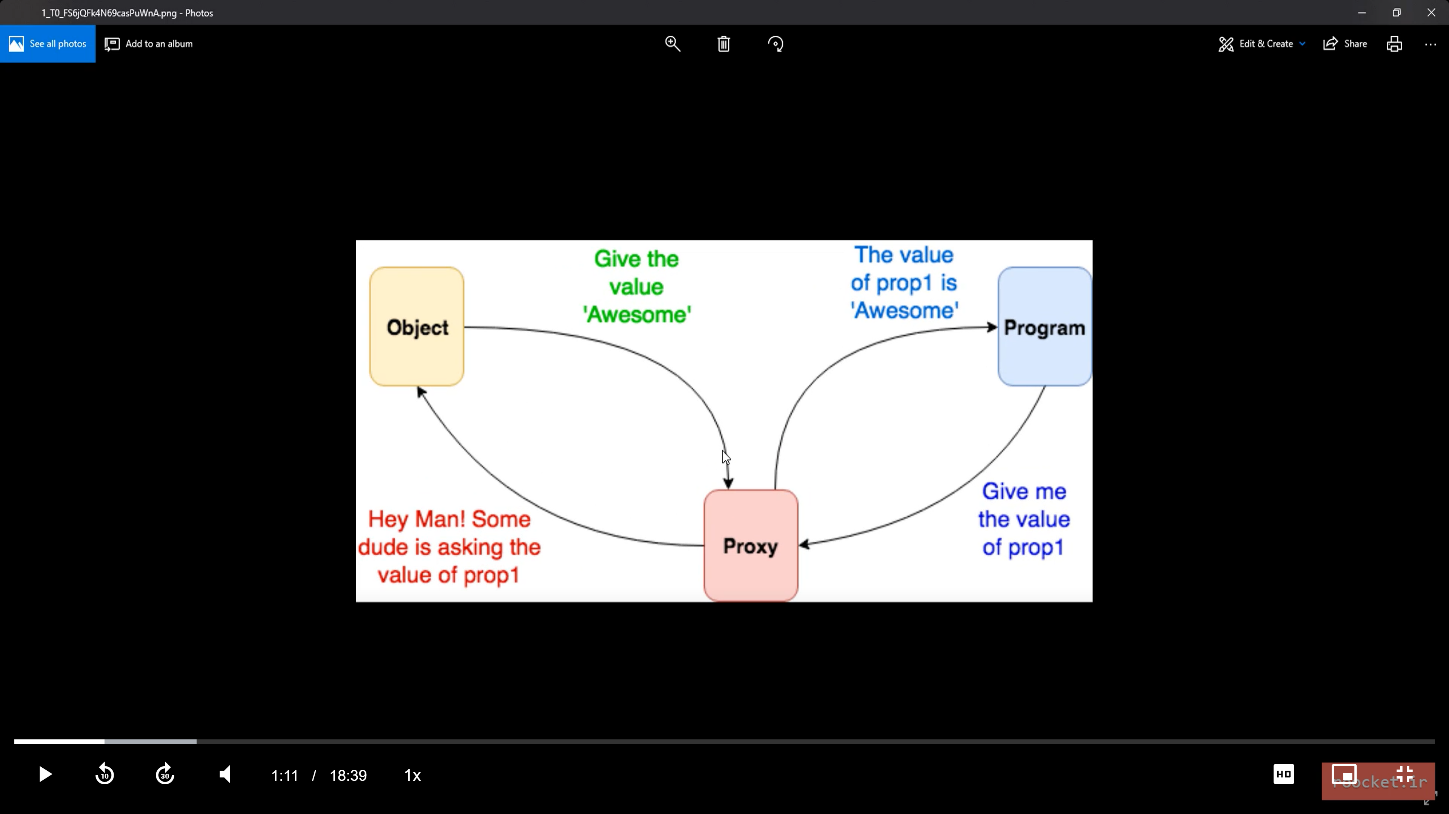
**Reflect API**

**Meta programming**

* It's a programming technique that every programing language implement in its specific way. It leads to a pack of tools that help to manage other code and tools. For example, in javascript, it can read, analyze, generate, modify, and manage objects of other classes. It's also more dynamic.
* For example, instanceof, typeof, and getter and setters in objects to manage the values are some tools that help with Meta programming.
* Reflect can not be created like an object, with the **new** keyword. It has some **static methods** that can be accessed.
* We had some of these methods before, but in Reflect they have some improved parts, and also they are packed as a set of (all-together) tools, easier to use.
* Reflect.**constructor**(constructor function or class, [array of properties], another object to inherit its prototype) // It's similar to the **new** keyword method but has its own professional usage.
* Reflect.**apply**(function to call, object that we want to access its properties with 'this' keyword, [array of other arguments of the function])
* Reflect.**getPrototypeOf**(obj)
* Reflect.**setPrototypeOf**(target object that we want to change its prototype, the object that target object wants to inherit its prototype) //Returns true if successful
* Reflect.**defineProperty**(target obj, keyname, **{**value : 'value', attributes**}**) // attributes for example be writable : false, enumerable, configurable.
* Reflect.**deleteProperty**(target obj, keyname)
* Reflect.**get**(target obj, keyname) // It can be used for arrays too, keyname would be indexes
* Reflect.**has**(target obj, keyname) //checkes if a property with that keyname exists
* Reflect.**set**(target obj, keyname, value) //If keyname exists it will replace its value.
* Reflect.**ownKeys**(target obj) // Returns an array of the keys. If it receives an array, it will return the length of that array.
* Reflect.**isExtensible**(obj) // Returns true if the obj can accept new properties. By default it's true for objects.
* Reflect.**preventExtensions**(obj)

**Proxy API**

* This is also about meta programming



* Let proxy = new Proxy(target object, handler)
* A handler is an object that can alter and define methods that work on the object.
* **get** method, that gets the object and the key as the parameters

class User {

constructor(name, lastName){

this.name = name;

this.lastName = lastName;

}}

let user = new User('Rosa', 'Johnson')

let handler = {

**get** (obj, propkey){

return propkey in obj ? obj[propkey] : "The key does not exist"

// return Reflect.has(obj, propkey) ? Reflect.get(obj, propkey) : "The key does not exist"

}}

let proxy = new Proxy(user, handler)

console.log(**proxy**.name)

console.log(Reflect.get(**proxy**, 'year'))

* **Has() in handler**

let handler = {

has(obj, propkey){

console.log("Do soemthing")

return Reflect.has(obj, propkey)

},

}

//proxy.has('name') //this does not work

console.log(Reflect.has(user, 'name')) // Direct access, true

console.log(Reflect.has(**proxy**, 'name')) // Through proxy, true

* **The set method in the handler**. We can define our own rules.

let handler = {

set(obj, propkey, value){

if (typeof(value) == "number"){ // only numbers

obj[propkey] = value

//Reflect.set(obj, propkey, value) //we can also use this

}}}

let proxy = new Proxy(user, handler)

Reflect.set(proxy, 'age', 24)

//proxy.has('name') // doesn't work; has is a static method in Object or Reflect

console.log(user.age)

console.log(proxy.age)

console.log(Reflect.has(user, 'age')) // Direct access, true

console.log(Reflect.has(proxy, 'age')) // Through proxy, true

* To access the object through its name, we can use **setPrototypeOf** to give the proxy to obj as a prototype.

let proxy = new Proxy({} , handler); // an empty target object, handler is like above

Reflect.setPrototypeOf(user , proxy)

carObj.year = 1234;

carObj.name = 'Derry';

console.log(user.name);

console.log(Reflect.has(user , 'year'));

* **Function in proxies**. // We can use proxies in functions too. Apply() for example. In the example below, we can use a proxy like a function because the sum is a function. Proxy()

function sum(a, b, …c) {

return a + b

}

let handler = {

apply(target, thisArg, args){

consol.log(` args are : ${ args }`)

return target(…args)

}

}

let proxy = new Proxy(sum , handler);

//proxy.apply(1,2)//this does not work proxy doesn’t have the apply

Proxy(1,2)

* **Proxy.revocable(obj , handler)** // To define a proxy to be revocable, which means the proxy becomes unusable. Setting a proxy as a prototype and to be revocable, returns two values which we have to separate, and then set the proxy part as the prototype, and use revoke to revoke the proxy.

let {**proxy**, **revoke**} = Proxy.revocable({} , handler);

Reflect.setPrototypeOf(user , proxy)

**revoke();**

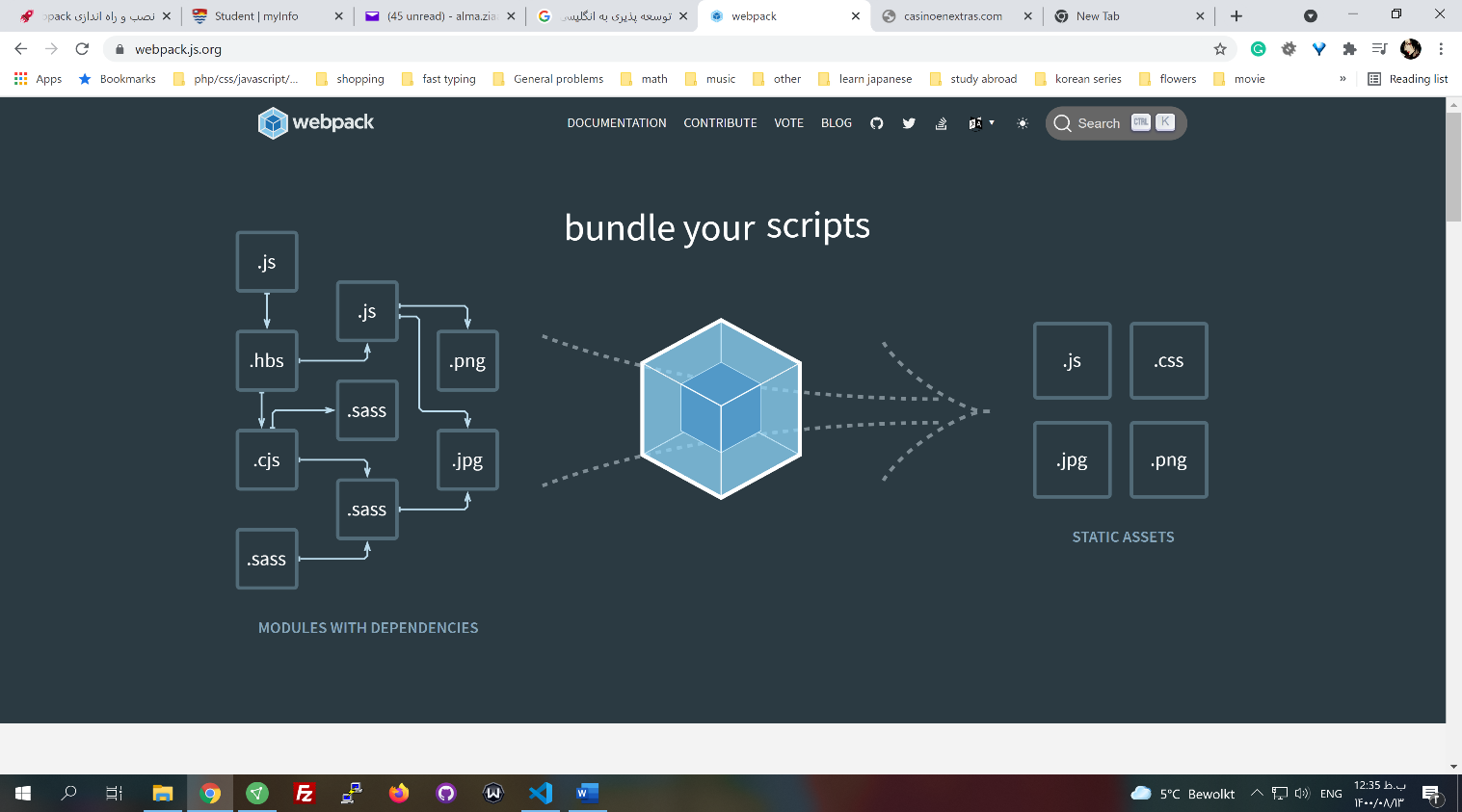
user.year = 1234;

**Modular**

* Before ecma6, you had to put all your codes in one JavaScript file and access it when you wanted. Putting all the codes in one file had some problems like readability, debugging, scalability, and ….
* Modularizing means that we put the codes that have a specific functionality in one file, like a class, and access it internally, in other JavaScript files. Before ecma6 we could use some packages to implement this behavior, but the packages were bulky and not practical. Like CommonJs, AMD, and Jadeja packages which had problems in actual usage.
* Putting JavaScript codes in different files and loading them in HTML, were not called modularizing. It only separated codes, and the order of the way you refer to them was important because some codes could refer to others.
* In NodeJs which is a platform used to run javascript as a backend language, modular concepts became important. It implanted a specific modular approach.
* Later in ecma6, it was implemented.
* It can't simply be used in browsers, a tool can be used to combine the modular codes during compilation. There were other tools in the past but **now** Webpack is a more common one.

**WebPack**

* It's a tool that has many functionalities. We can refer to many formats in our codes and webpack compile them to a few files known to browsers. To have it we should install npm package manager. To install npm we can install nodejs.



* npm init. It creates a file in that project that manages packages.
* npm install webpack webpack-cli
* a folder named **node\_moduels** is created which contains native node codes and all the extra packages downloaded from nod.js.org
* Create a file named **webpack.config.js** with this content:

module.exports = {

context: \_\_dirname + "/src",

entry: "./script.js",

output: {

path: \_\_dirname + "/build",

filename: "bundle.js"

},

module: {

}

}

* Create a folder named src and put all the js files inside that. Entry is the main file that imports other files and we want to run.
* in **package.json** inside the script section we write:

"build": "webpack-cli --mode=production", // production compresses the codes

"watch": "webpack-cli --mode= development --watch" //to automate the run as files change

* npm run build //runs the script in package.json and webpack-cli will read the webpack.config.js file
* npm run watch
* it creates a folder named build and a file named bundle which contains our codes and some extra codes
* export default X
* import X from "./Y"
* In the HTML file, change your referred js file to the new created bundle file.
* npm i //If we have package.json file, that has the dependencies, we can only run this command, it will download the packages
* export { } // to use modules of a file in another js file, those objects, functions, and variables should be exported. We can use different methods to export items, but a usual way is to use an object.

function sayHi(){

return "Hi"

}

let obj = { 'name': 'Derry'}

let age = 35

export {

obj,

age,

sayHi

}

* Import { } from "file address" // Inside the files we need to import items. With **obj destructuring,** we can put them into different items. The name of the new variables should be the same as their exported ones. **As** keyword is used to change the names, instead of colons that we had in normal destructuring.

import **{**obj as name, age, sayHi**}** from "./logger" //using obj destructuring

* **export default item** // To import an item without destructuring. It can be used only once in a file. Using this method, the name of the item can be changed when it is imported.
  + In export file

export {

item

}

export default new Users() //export an object

* + In import file

import **user1, {item}** from "./logger" //user1 is the default

* \* is used to import all the items at once.

import **\* as log** from "./logger" //returns a module object containing items

let user1 = log.default // in this method default exported item is called default

console.log(user1.name)

* To export a variable directly the format below with const is always used.

**export const item** = function() {} // Can't be used for export default

**Babel Js**

* Babel Js is a javascript compiler. It compiles new javascript codes to a form that is compatible to older versions of browsers.
* We can use wepack to install it. Check the website for latest installation guid.

npm install --save-dev babel-loader @babel/core

* Copy this code to webpack.config.js file inside the module section

rules: [

{

test: /\.m?js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

presets: ['@babel/preset-env']

}

}

}

]

* Create a file with the extenstion .babel.config.json and run the code
  + npm install @babel/preset-env --save-dev

copy this into the file

{

"presets": ["@babel/preset-env"]

}

* npm run watch