ECMAScript 6 (2015)

* ECMAScript is a language that JavaScript is made based on it. 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 specific 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 we can change the items if it's an object or an array.

const NAMES = { age : 23 };

NAMES.age = 20;// ok

NAMES = { } //not ok, value changed

NAMES.name = 'Derry' //ok

console.log(NAMES);

* **||** // to assign variables we can use this operand, it checks if the first one is not false, for example, if the item does not exist (item +1) is NaN and is considered false and It assigns the second 1 to the item.

Let item = item + 1 || 1

Arrow functions

[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, you can make it even shorter by removing braces and the return keyword

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

* If the returned 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 item, we can remove the parentheses

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

* **This** keyword here points to the person's obj. Suppose you have a constructor function that uses an anonymous function somewhere. This anonymous function needs the constructor's properties. In that case, you can declare a global variable assigned with **this** and use it. In ecam6, you can use **arrow functions** instead.

function Person() {

this.age = 0;

**that** = **this**;

setInterval(function(){

that.age++; //this refers to setIntervals

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 : **() =>** {

// this.a = ""

console.log(this.a , this)

}}

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

obj.c()//a is undefined, this is the global variable, which means it refers to the **c function itself**

* **arguments[index]** // is an object's global variable that you can access inside every function declared by the function keyword. Argument[1] shows the first argument of that function. **It's not accessible inside an arrow function.**
* You can't use Arrow functions as constructor functions, and **they don't have any prototypes**.
* If you call an arrow function in an eventListenrer that is declared after the eventListenrer, you may face an error, which doesn't happen if you use the function keyword.

Function default values for 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 you don't know how many arguments you may have, you could use arguments[index] to access them in normal functions. But what about arrow functions?
* ... c says the rest of the arguments, no matter how many, are 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 , []

}

* The rest should be at the end. You can't have another item after it.

Spread … list to items

* Before ecma6, you could use **apply** to pass the items of a list as separate parameters.

function sum (a, b, c ) {

console.log(a + b + c)

}

var list = [1, 3, 5]

sum.**apply** (**null**, list)

* Spread separates items of a list // Rest parameter syntax will create an array from an indefinite number of values.

function sum (a, b, c ) {

console.log(a + b + c)

}

let list = [1, 3, 5]

sum(... list)

* You can use it to merge lists

let list1 = [1, 3, 5]

let list2 = [0, 7, 5]

let list3 = [... **list1,** ... **list2**]

For-of

* You can use it to iterate through **iterables**, like lists, strings, maps, sets, and generators. **Object** is not iterable. The difference with for-in is that here we can get the item itself while in for-in we could access the index, and also for-in can be used for objects too.

let list = [1, 3, 5]

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

console.log(item)

}

Destructuring

* Used in arrays and objects
* Destructuring is used to create variables from array items or object properties.
* To easily assign items of arrays and objects into separate variables

**Arrays Destructuring [ ]**

let list = [1, 3, 5]

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

* to assign values of multiple variables in one line

let a , b

[a , b] = [1 , 2]

* Default values

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

* You can use it to separate items of a list returned from a function
* **Ignore** some items

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

[a , b] = [15 , 7]

console.log(a , b)

let list = [1, 3, 5]

let [item1, , item2] = list // The second item is ignored, the value of item 2 will be 5

* You can usethe **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)

* Using variables in strings and joining them together was very troublesome before ecma6. We had to use +, and that would be messy in long strings with too many variables.
* **``** // backticks are called template literals. You can code in any number of lines with breaks; that was not possible with double quotes.
* To use variables with strings, you can use `Hello` **+** var, but there is a better solution:
  + **`** string **${**var or mathematic calcualtions**}** string **`**

let obj = {

name : 'Derry'

}

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

Goodby`)

* You can use it with short-ifs

Class

* You can use classes to create and manage new objects and support inheritance—a key ingredient in object-oriented programming and a mechanism for **reusing code**.
* Classes in JavaScript are made by constructor functions; they only 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 car = new Car( "Benz");

car.run();

* To access this in an eventListener in a method of a class, use arrow functions.

class draggable {

constructor(el) {

this. element = el

addDnDHandlers()

}

addDnDHandlers() {

//code

this.element.addEventListener('dragstart',this.handleDragStart)

//this.element.addEventListener('dragstart',**(e)=>this.handleDragStart(e)**)

//this.element.addEventListener('dragstart',**this.handleDragStart.bind(this)**) }

handleDragStart(e) {

console.log(**this**);// with the first method can't access **class this**

console.log('drag start' , e.target);

}

}

* You can define properties both of these ways in a class
  + First, define the property (without the need for the let keyword) and then assign it, It is more readable

class Car {

**model;**

**year;**

**original;**

constructor(model, year){

this.model = model

this.year = year

}

run(){

this.original = true

console.log(this.model);

}

}

let car = new Car( "Benz");

car.run();

* + This is similar to the above and with **this,** the properties are set for the class

class Car {

constructor(model, year){

this.model = model

this.year = year

}

run(){

this.original = true

console.log(this.model);

}

}

let car = new Car( "Benz");

car.run();

Inheritance In Class

* To use the properties of a mostly similar class
* Y extends X //keyword
* super // Refers to the parent class. You can use it to access properties and methods in the parent class. When you rewrite a property but still need the parent one, you can also use it.

**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 car = new BMW ( "Benz", 1989, "blue");

car.run();

* **This** keyword in inheritance: With **this** keyword, we also have access to the methods and properties of the parent class. Using this before calling super is not allowed.

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

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

//**this**.model = model// we can’t use **this** before using super

**super(model)**

this.color = color

}

fly(){

**this**.run(); //here we can access the parent properties by **this**

}

}

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

car.fly();

Static methods

* To access a class method without having a **new** object. You can declare those methods as static methods. You **cannot** access static methods through objects.

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 car = **new** BMW ( "Benz", 1989, "blue");

**car.run()**; // It runs 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 the purpose of indirect access to properties. Also, you can use them 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, you can access that property 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** an actual property value, and you can only access it in this way.
* get keyword // to create a method as a getter. An **object** can access this method like a **property** to **get** an actual property value, and you can only access it 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]) //return the last value

}

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:**

* We don't need to write the name of var if it's the same as a property
* **[keyname1]** // variables as property name
* **"say hello"()** // string as the name of method

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, the second object has priority.

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

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

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

* In objects made out of classes, it works the same, and the prototype of the returned object is the Class of the **target object**.
* Object.assign({}, any number of objects) // target is a normal empty object. You can use it to prevent the change in the 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

}}

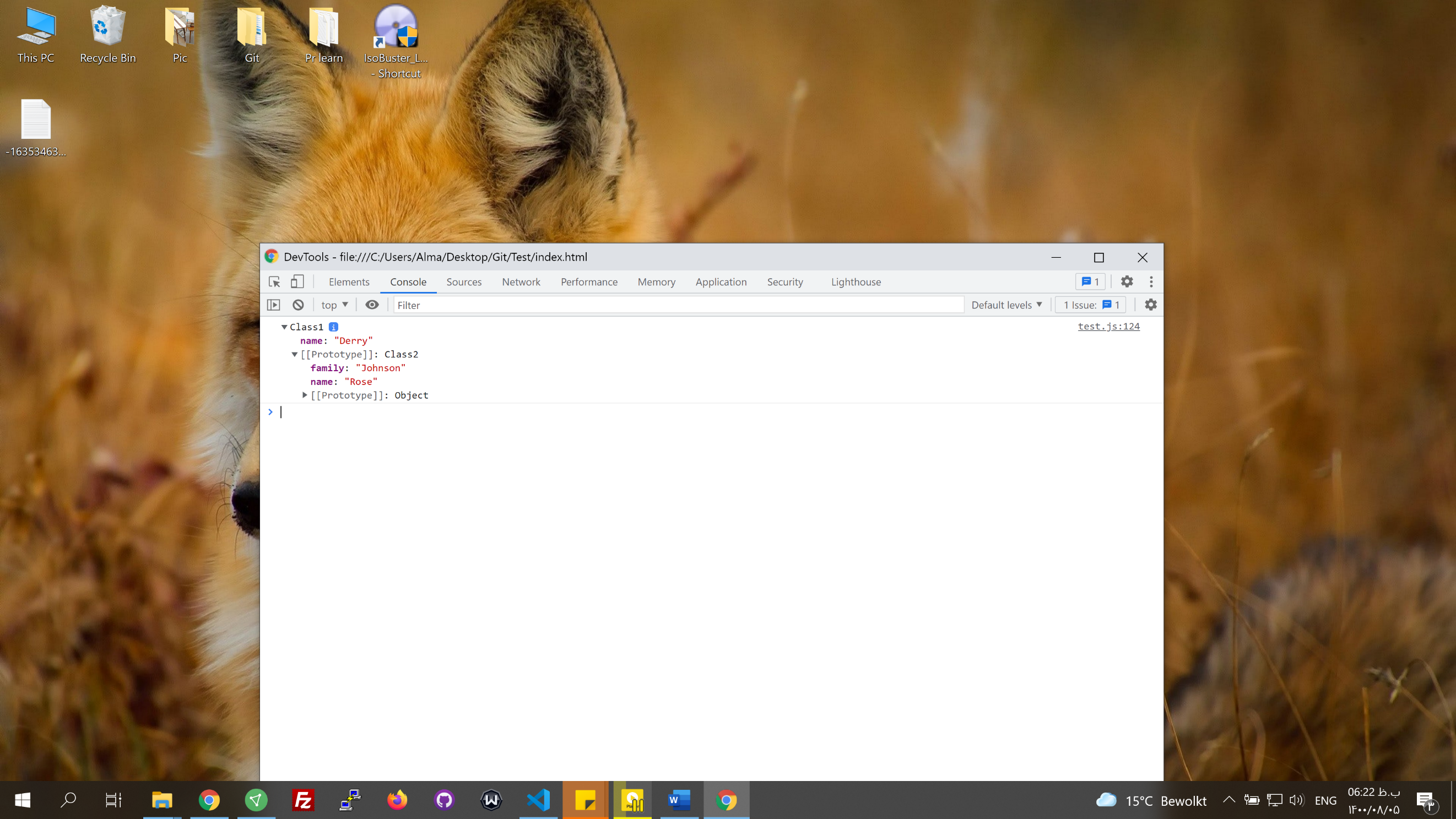
let obj1 = new Class1();

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 is, but it's always a new unique number
* We don't need the **new** keyword to create a symbol like what we do with string constructor functions. (new String())
* You can use them 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
* You cannot access the unique number directly.
* They are hidden in loops like for-in.
* To access them, you 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 is shared
* Symbol.for('description') // creates the symbol and under that description stores it somewhere. We use exactly this to access it anywhere.

**Well-known Symbols**

* They are pre-defined symbols that are built-in in JavaScript, which are like ids, that you use to change the behavior of some built-in methods.
* Symbol.replace // with this symbol; you can completely change the replace method behavior.

class **ReplaceX** {

constructor(value) {

this.value = value;

}

**[Symbol.replace]**(string) {

return `/${string}/${this.value}`;// returns this when you pass an object of this constructor as a property to **replace(obj)**

}

}

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

console.log('foo1'.**replace**(**new ReplaceX('bar')**)) // output: /foo1/bar

Arrays new methods

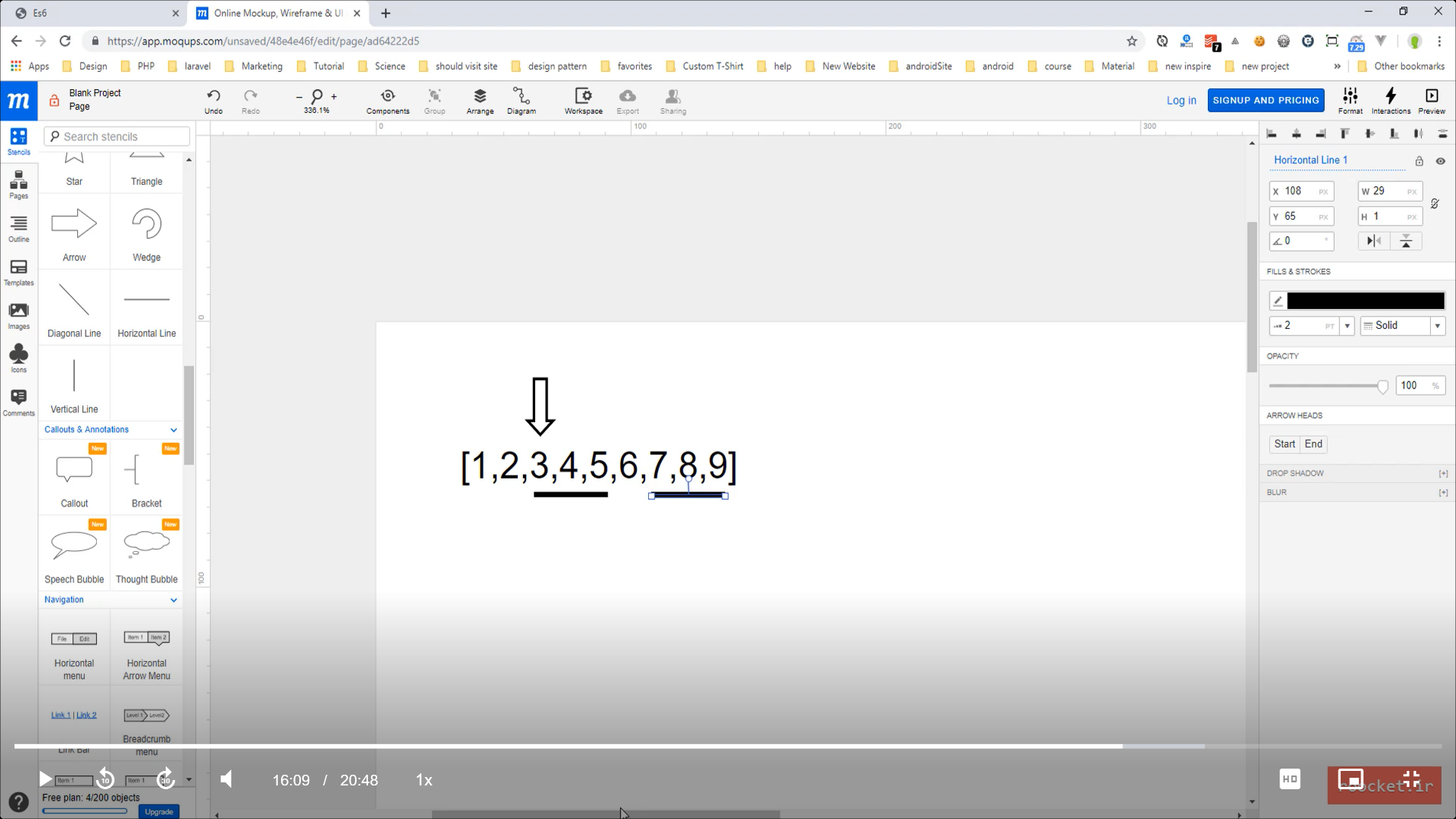
* **Array**.**of**(items) // If we use **Array**(2), it returns an empty list instead of a list holding 2. It returns a list of undefined items, and the length is the number given. It happens only with numbers and only one number. This problem is solved by **of** constructor function, which is a **static method of Array**.
* **Array**.**from**(list, item => item\*2) // Is a static method. The second parameter is a map method. It elevates readability.
* list.**find**(item => item % 2 == 0) // the difference with the filter is that the filter returns all the items that match the condition while finding returns the first one that matches and exits. It returns the found item, not in a new array.
* list.**entries**() // returns an iterator holding a group of separate lists of one item and its index. You can use it to traverse the lists without the use of loops like for-of, with **next().**

let list = [5,9,3]

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

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

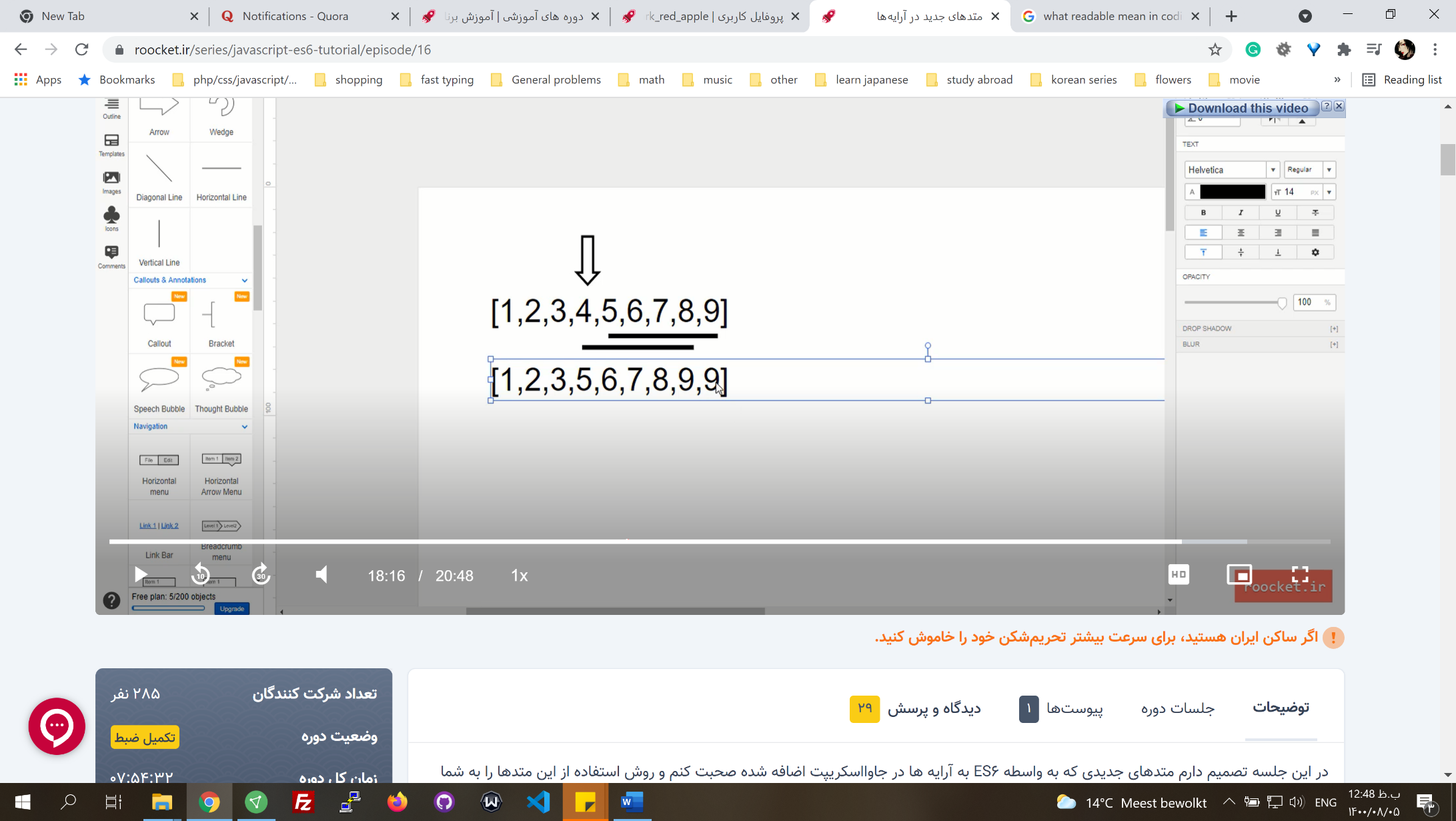
* list.**fill**(item,startindex,endindex) // it fills(replaces) all the items of **list** with the new item. Also, you can say what indexes you want to fill. The item with endindex is not affected.
* list.**copyWithin**(targeindex,copystartindex,copyendinex) // is used in big arrays (buffers) to replace items with other existing items in the list. If you don't give the endindex it copies to the end. The item with endindex is not included.



Copy this

To this

Target



Numbers new methods

* **IsNaN**(var) // 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.**trunc**() // Math.Floor rounds down, Math.Ceiling rounds up, and Math.Truncate rounds toward zero. Thus, Math.Truncate is like Math.Floor for positive numbers, and like Math.Ceiling for negative numbers.

Strings new methods

* 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. Returns true or false.
* str.**startsWith**(string, position) // It checks if from giving the position a string starts, not in the whole string.

'Derry'.startsWith('rry', 2)

* 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: done, value, and the next value in the iteration sequence.
* **Symbol.iterator** // All the iterables have this method, which is a well-known function you can access by a symbol. You could access symbols by [ ].
* You can use this method in **for-of loops**. It calls this method behind the scene, runs the next() function, and gives the value to the for-of variable. 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]()) // Array Iterator

for(let item of list) {

consol.log(item)

}

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

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 has 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**;

**}** }}

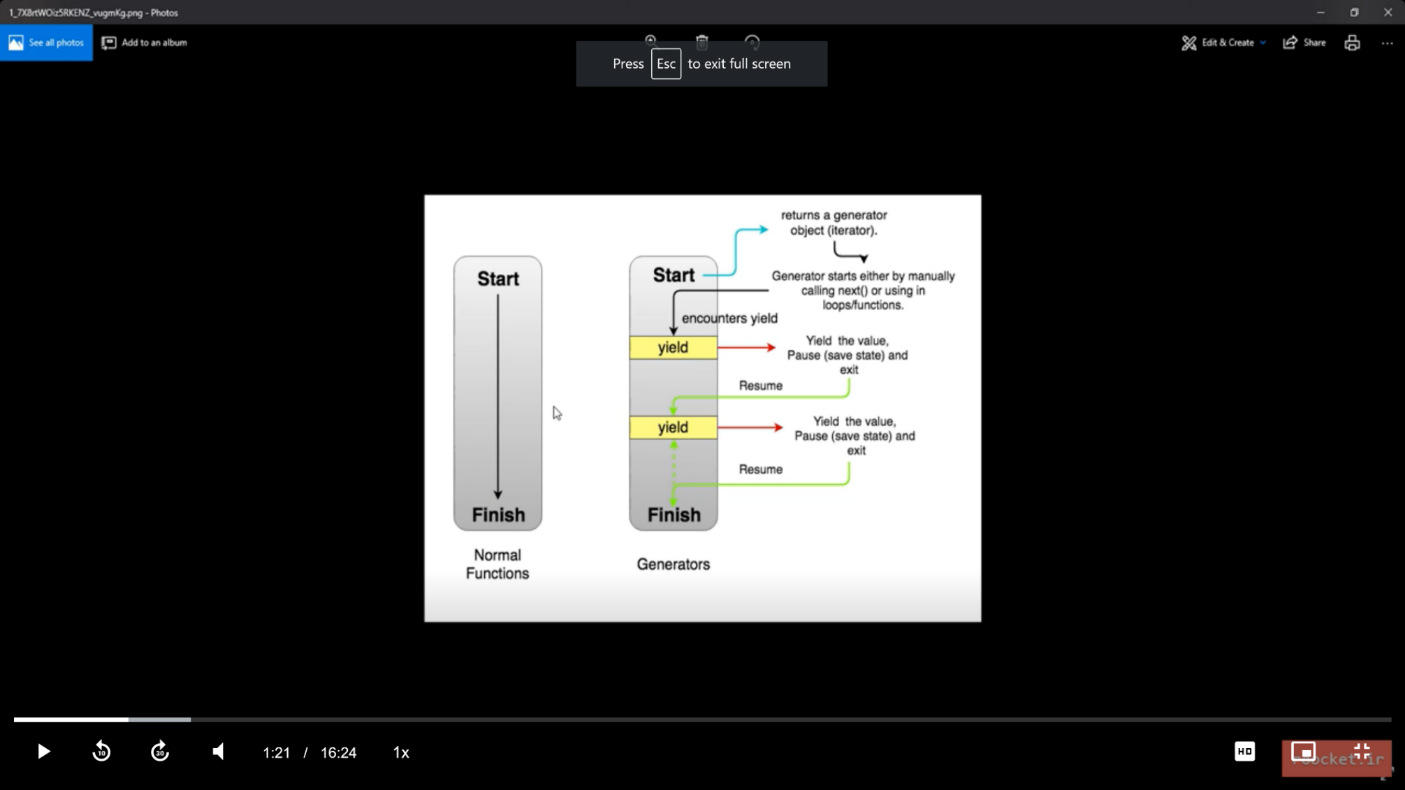
for(let post of user) {

console.log(post);

}

Generators

* Generators are a kind of function that can stop and continue at specific points. We mark these points by the keyword **yield**.
* You can use them when you need a value of that function at some point in the program, and then you need the next one at another forward point.
* When you declare a generator, an iterator is also assigned to it. This iterator is returned from this function and is what enables the stop and continue process. You can call next(). Next() works based on yield points to run the code partly. Yield points can return a value.



* To declare a generator, if :
  + It is a normal function put a **\*** after the 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 runs this the done value is true.}

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

console.log(namesIt.next())

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

console.log(name)

}

console.log(**[**... namesIt**]**) // use the spread opertaor to assign yield values to a list. Runs only one time.

* In objects, you can use them instead of iterators to write cleaner and shorter codes. The iterator example above 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 downward. An action is considered asynchronous if it needs time to be done, like a connection to a server or a database. 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 the return doesn't work correctly**.

function doSomething() {

setTimeout(function(){ // a simulation for time consuming jobs

**return 'code 2'**

},2000)

}

console.log('code 1')

console.log(**doSomething()**)// it won't show code 2

console.log('code 3')

* Callbacks handle these returned values. We pass a function (called callback) as a **parameter** to the original function, then it is called whenever needed. You can pass data to the callback.

function doSomething(**callback**) {

setTimeout(function(){

**callback('code 2')**

},2000)

}

console.log('code 1')

doSomething(**funciton(data**){

**console.log(data)**

})

console.log('code 3')

* Since callbacks are executed asynchronously, they become a hell of a nuisance to handle, regarding 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 function names and don't declare them as a parameter, and use function definition instead of assignments.
  + The usual bad way:

let num = 1

multiply(num, **function**(multiplyData) {

minus(multiplyData, **function**(minusData){

//do something

let data = minusData + 4

return data

})})

function multiply(num, callback){

//do something

let multiplyData = num \* 5;

callback(multiplyData)}

function minus(multiplyData, callback){

//do something

let minusData = multiplyData - 2;

let outp = callback(minusData);

console.log(outp)}

* + To improve, declare callbacks with **function definitions** and with **names**:

multiply(num, multiplyCallback)

**function** **multiplyCallback**(dataMultiply){

//do something

minus(dataMultiply, minusCallback)

}

**function** **minusCallback**(dataMinus){

//do something

let data = dataMinus + 4

return data

}

Promise

* Promises are easy to manage while dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.
* They are used to remedy the problem of the chain of callbacks. It promises that retunes the data and handle the error correctly.
* Instead of writing a confusing chain of callbacks and handling their errors separately, a chain of promises improves **readability** and **simplifies** the way all the **returned data and errors** in this chain are handled.
* new **Promise**((**resolve**, **reject**) => { resolve(data); reject(error) }
* func.**then**(data => { }, err => { }).**then**().**then**() // multiple thens to handle data for different promises returned from the previous then.
* func.**catch**(err => { }) // handle 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) //like calling callebacks

})

}

function func2() {

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

//code

**reject**(error)

//code

**resolve**(data)

})

}

func()

.**then**(data => **{**func2(data).**then**(data2 => { //then inside then, here because of **{}** in arrow function, there is no return for the first then

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 on the same level to work on the data returned from the second promise.

func() //func returns a **promise (1)**

.**then**(data => func2(data))//then is a method in func that returns a **promise (2)** by func2 which is returned implicitly by arrow function

.**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**([x,y,z]) // To perform promises together and return their data in a list. It accepts an array of promises or non-promises. It performs all the promises. If **one promise is rejected**, it always goes into the **rejected mode**, no matter if all other promises are resolved. 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 rejected !!

Promise.**race**([promise1, promise3])

.**then**(data => console.log(`race ${data}`)) // it returns race resolved !!

.catch(err => console.log(`race ${err}`))

Map

* A collection data structure that enables **key-value** format.
* let map = new Map()
* map.**set**(key,value) // key can be a string, number, symbol, or 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**) //access by the same variable

* map.**size** // the number of items
* map.**delete**(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) { //destructuring []

consol.log(key,value)

}

* map.**keys**()
* map.**value**()
* map.**entries**() //It returns an iterator that you can loop through with the next()
* Convert lists to maps. 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**(item)
* 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, spread
* [… mySet] // Convert to a list

let mySet = new Set(['item1', 'item2', 'item3', 'item1'])

console.log(mySet)

console.log([... mySet])

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 **Map**() //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!

**WeakSets**

* 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**

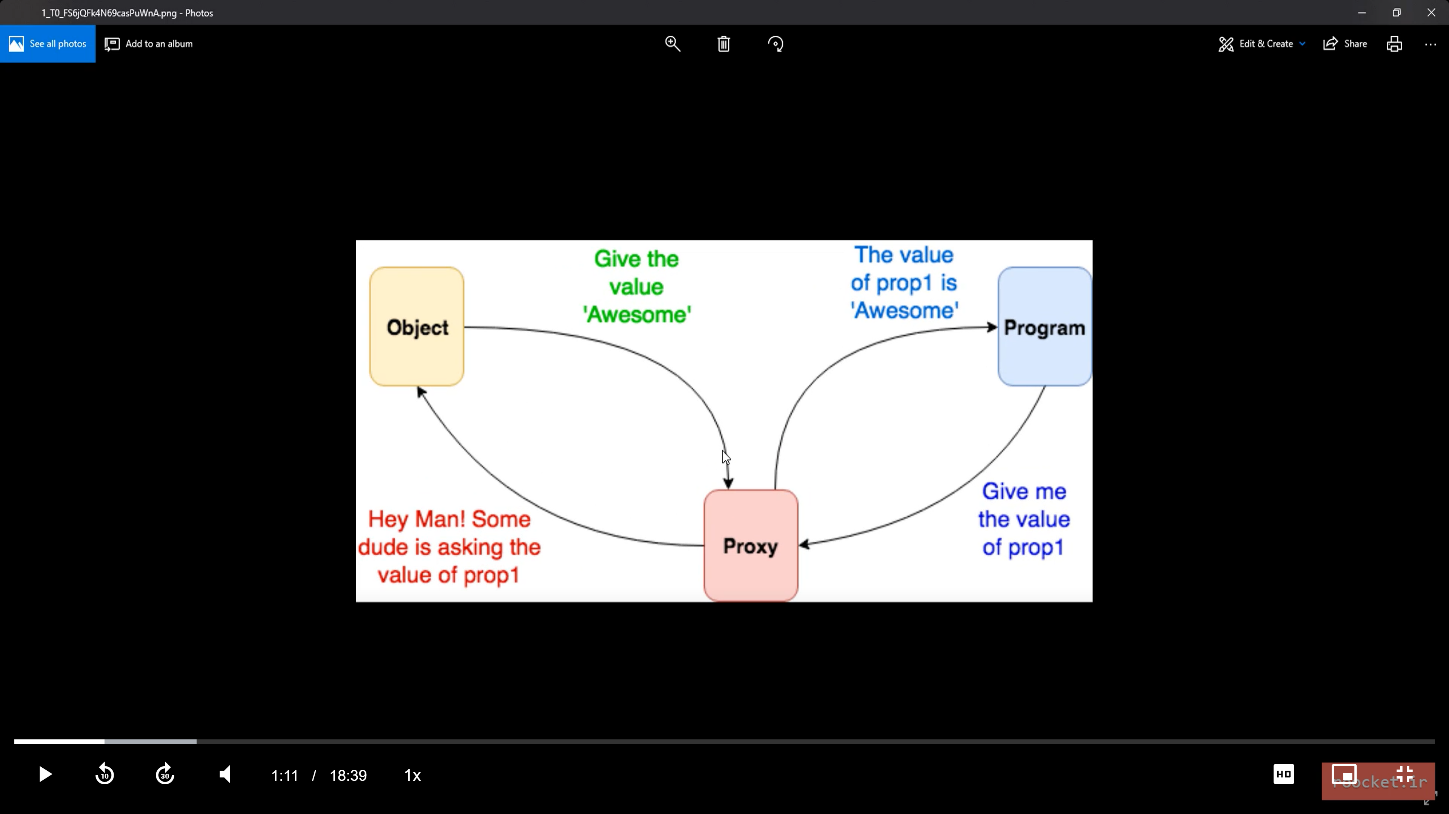
* It's a programming technique that every programing language implements in its specific way. It leads to a **pack of tools** that help to **manage other codes and tools**. For example, in javascript, it can read, analyze, generate, modify, and manage objects of other classes. It's also more **dynamic**.
* For example, **instanceof**, **type of**, and **getter and setters** in objects to manage the values are some tools that help with Metaprogramming.
* Reflect can't 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**.

**Methods**:

* **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, [other arguments of the function as an array])
* **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. Keynames are 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 returns 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 metaprogramming



* Let proxy = new **Proxy**(target object, **handler**)
* A handler is an **object** that can alter and define methods that work on the object. For example **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('Rose', '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 something")

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 //you can't use obj.propkey because obj is the proxy not user

//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 directly through its name but as a proxy, we can use **setPrototypeOf** to set that proxy as a prototype.

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

Reflect.**setPrototypeOf**(**user** , **proxy**)

user.year = 1234;

user.name = 'Derry';

console.log(user.name);

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

* **Proxies for functions**. // You can use proxies for functions too. Apply() for example. In the example below, you can use a proxy like a function, proxy(1, 2), because the sum is a function.

function **sum**(a, b, ...c) {

return a + b

}

let handler = {

**apply**(target, thisArg, args){ // here we use apply only to pass the args to sum

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

console.log(thisArg)

return target(...args)

}

}

let proxy = new Proxy(**sum** , handler);

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

**proxy(1,2)**

* **Proxy.revocable(obj , handler)** // To define a proxy as revocable, that means it becomes **unusable**. To have a revocable proxy as a prototype, use the **revocable()** **static method**. It returns two values; you should separate them by object destructuring, set the proxy part as the prototype, and use the revoke to revoke the proxy.

let {**proxy, revoke**} = Proxy.**revocable**({} , handler); // returns an object{}

Reflect.**setPrototypeOf**(user , **proxy**)

**revoke();**

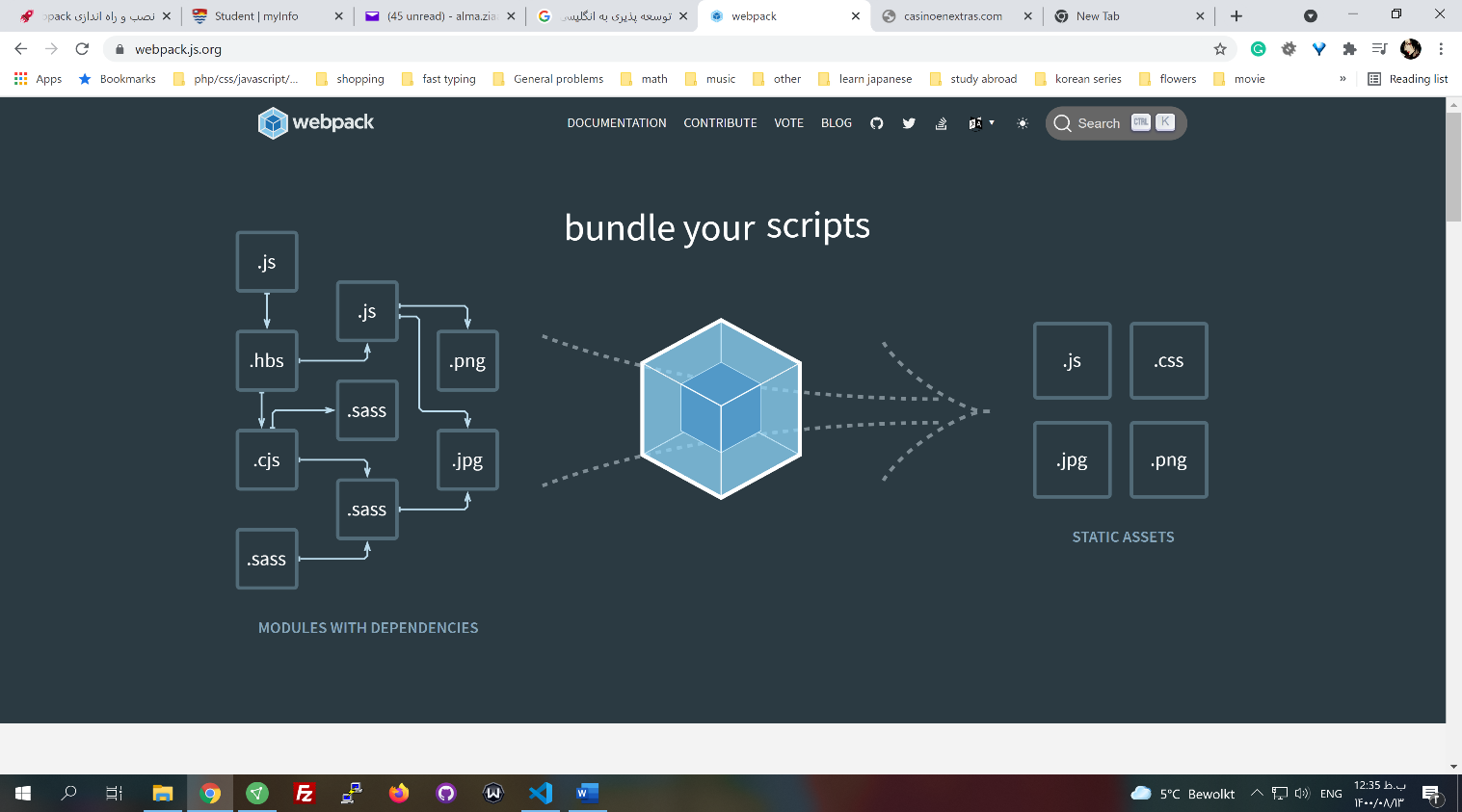
user.year = 1234;

Modular

* Before ecma6, you had to put all of your codes in one JavaScript file and call it when you wanted. Putting all the codes in one file had some problems with **readability**, **debugging**, **scalability**, and ….
* Modularizing means putting the **codes with specific functionality in one file**, like a class, and accessing 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, they had practical problems.
* Putting JavaScript codes in **different files and loading them in HTML** were not called modularizing. It separates codes. Also, the order of the references was important because some codes would need others.
* In **NodeJs**, a platform used to run javascript as a back-end language, modular concepts became important. It implemented a specific modular approach.
* Later in ecma6, **import,** and **export** were introduced.
* **You can't simply use it in browsers**. You can use a tool 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 use many formats in our codes that webpack compiles to a few files known to browsers. To use Webpack, we should install the **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 called **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 starter and the main file that imports other javascript files.
* 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 has our codes and some extra codes
* npm i //If you have the package.json in your project, it contains the dependencies it needs, so you can just run this command and it will download the packages it needs.

**Export/Import**

* export default X
* import X from "./Y"
* In the HTML file, change your referred js file to the created bundle file.
* export { } // If you want to use objects, functions, and variables of a file somewhere else you should use export in the file they are written. We can use different methods to export items, but the 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" // To import items. With **obj destructuring,** put them into different variables. The name of imported variables should be the same as their exported ones. You can use **as** to change the names instead of colons in normal destructuring.

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

* export default item // To import an item without destructuring. You can use it 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)

* The format below with const is always used to export a variable directly.

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

Babel Js

* Babel Js is a javascript compiler. It compiles newer javascript codes to a form compatible with **older versions of browsers**.
* You can use Webpack to install it. Check their website for the latest installation guide.

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

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

module: {

rules: [

{

test: /\.m?js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader",

options: {

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

}

}

}

]

}

* Create a file with the extension **.babel.config.json** and run this code
  + npm install @babel/preset-env --save-dev
  + copy this into the above file

{

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

}

* npm run watch
* You can use a browser list to specify what versions of browsers you want to include. Check @babel/preset-env documentation.
  + You can use queries to specify the level of compatibility.

{

"presets": [

[

"@babel/preset-env",

{

**"target": "> 0.1%"**

}

]

]

}