جلسه دوم – تعریف متغیر با Const و Let

در ES6 همانگونه که در دوره جاوا اسکریپت نیز گفته شد به جای var برای تعریف متغیر از let استفاده می کنیم.

عمده ی مشکلات var مربوط به scope است مثلا در function یا در حلقه از var که استفاده می کنیم در خارج از scope آنها متغیر همچنان در دسترس است و در function و کلا می توان دو بار متغیری را تعریف کرد.

ساخت متغیر با استفاده از const

Identifier متغیر های از نوع const را تماماً با حروف بزرگ می نویسند البته می توان اینکار را نکرد اما پیشنهاد میشود رعایت شود.

متغیر های از نوع const یا ثابت ها مقدار دهی که میشوند دیگر قابل تغییر نیستند و اگر سعی در تغییر دادن مقدارشان بکنیم error برگشت داده خواهد شد.

البته اگر ثابت حاوی لیست باشد می تواند روی آن push داشته باشیم اما در ثابت نمی توان مقدار را آپدیت کرد. این داستان برای آبجکت ها نیز صادق است آبجکت قابل تغییر نیست اما پراپرتی ها را میشود تغییر داد.

در فریم ورک ها و پکیج ها از let استفاده شده است غالباً.

جلسه سوم – توابع arrow

در توابع arrow دیگر نوشتن function لازم نیست:

let sum = (a, b) => {

return a + b

}

در arrow function اگر فقط یک خط کد را در بلاک function داریم می توانیم return و { } را برداریم و مستقیم آن یک خط کد را در ادامه => قرار دهیم:

let sum = (a, b) => a + b

نیتجه تابع فوق همانند تابع قبلی است. Arrow function ها در ساده سازی کد نویسی بسیار کاربردی هستند. در حالتی که از { } استفاده می کنیم اگر در برگیرنده ی آبجکت باشد برای اینکه آبجکت مه با { } محصور شده است را با { } خود اشتباه نگیرد به جای استفاده از { } از ( ) پرانتز استفاده می کنیم برای بلاک کد تابع.

نکته: در ES6 اگر بخواهیم مقداری را درون آبجکت درون یک key بریزیم اگر key و identifier ایی که می خواهیم به key انتساب دهیم باهم هم نام باشند فقط که را می نویسیم.

قبل:

let name = ‘aran abdi’

let obj = {

name: name

}

بعد:

let name = ‘aran abdi’

let obj = {

name

}

کاربرد arrow function در .map

arr.map( (item) => item \* 2 )

البته اگر بلاک کد حاوی بیش از یک خط کد باشد نوشتن { } ضروری است.

حتی اگر arrow function فقط یک پارامتر داشته باشد می توان بصورت زیر آن را نوشت:

arr.map( item => item \* 2 )

کاربرد arrow function در توابع سازنده یا construction function

وقتی در تابع از this استفاده میشد باید برای در دسترس بودن this و اشاره کردن آن به شی از bind، call و apply استفاده می کردیم. اما در arrow function کلمه کلیدی this به متد سازنده اشاره می کند. البته برای حالت قبل می توانستیم از let that برای اشاره به شی استفاده کنیم اما راه حل جالبی نیست.

function Person() {

this.age = 0

setInterval( () => {

this.age++

},1000)

}

در کد فوق this.age که درون arrow function قرار گرفته به پراپرتی age در Person اشاره می کند اما اگر از function استفاده می کردیم this به خود تابع اشاره می کرد.

در موزیلا موارد بیشتری در مورد arrow function ها وجود دارد که پیشنهاد می شود مطالعه گردد.

جلسه چهارم – توابع arrow – بخش دوم

دسترسی به کل آرگومان های پاس داده شده به تابع

function sum (a, b, c, d) {

console.log(arguments)

}

در کد فوق arguments لیستی از آرگومان های ورودی یک تابع را برمی گرداند و متغیر ذخیره شده ای است که در داخل همه ی توابع وجود دارد.

اگر در خارج از تابع متغیری به نام arguments داشته باشیم در داخل تابع در دسترس نیست و کلاً پیشنهاد میشود نام متغیر را برابر با عبارات رزرو شده توسط خود جاوا اسکریپت قرار ندهید.

اما در arrow function ها حالت در شرایط فوق arguments ایی که سراسری تعریف شده است در دسترس است چون توابع arrow این مقدار را در خود ندارند.

arrow function نمی تواند تابع سازنده یا constructor function باشد و به prototype دسترسی ندارد و prototype ندارد.

اگر متد داخل آبجکت را به استفاده از arrow function تعریف کنیم به this به خود متد اشاره می کند نه به شی.

let obj = {

a: 10,

c: () => {

console.log(this.a)

}

}

کد فوق به خطا بر می خورد و خطا می گوید که this.a تعریف نشده است چراکه همانطور که گفتیم this به خود متد اشاره می کند نه شی.

جلسه پنجم – پارامتر های پیش فرض و عملگر Rest

در ES5 پارامتر های یک تابع نمی توانستند دلخواه باشند و برای دلخواه کردن آن ها از کد زیر استفاده می کردیم:

Function sum(x, y) {

let a = typeof x === ‘undefiend’ ? 1 : x

}

یا

Function sum(x, y) {

let a = x || 1

}

اما در ES6 در خود هسته برای این شرایط حالتی در نظر گرفته شده است:

Function sum(x=1, y=1) { }

در کد فوق برای x و y حالت پیش فرض در نظر گرفته شده است و دیگر ضروری نیست که مقداری به تابع پاس داده شود. توجه کنید که پارامتر های دلخواه باید پس از پارامتر های ضروری نوشته شوند.

عملگر Rest

مواردی که توضیح داده میشوند برای توابع عادی و arrow یکسان است.

برای گرفتن آرگومان های پاس داده شده به تابع از arguments استفاده می کردیم اما حال می توان از عملگر rest استفاده کرد:

function sum(a, b, …c) { }

در تابع فوق دو آرگومان اول به ترتیب به پارامتر a و b داده میشوند و بعد از آن هر تعداد آرگومان به تابع پاس داده شود در c ریخته میشود و c لیستی از آرگومان ها خواهد بود.

مثال:

function sum(..args) {

return args.reduce( (pre, current) => {

return pre + current

})

}

دستور فوق همه ی آرگومان های پاس داده شده به sum را جمع می کند.

حالت کوتاه برای نوشتن if و else:

let result = CONDITION ? IF TRUE : IF FALSE

جلسه ششم – آشنایی با سینتکس Spread

در ES5 برای پاس دادن تک تک اعضای یک آرایه یا لیست به یک تابع باید بصورت زیر عمل می کردیم: sum.apply(null,list)

اما در ES6 از Spread استفاده می کنیم:

console.log(…list) 🡺 تک تک اعضا را نمایش می دهد

sum(…list) 🡺 تک تک اعضا را به تابع پاس می دهد

همچنین می توان از آن برای یکی کردن دو تا لیست استفاده کرد:

list1 = [4, 5, 6]

list2 = [1, 2, 3]

list3 = […list1, …list2]

هر دو لیست در یک لیست با همدیگر ادغام می شوند.

جلسه هفتم – پیمایش حلقه ها با For-of

در ES5 برای پیمایش از .forEach استفاده می کردیم اما در ES6 حالت جدیدی به نام For-of داریم:

let list = [..]

for (let item of list) {

}

آبجت ها قابل پیمایش نیستند چه در این حالت و چه در .forEach اما هرچیز دیگری که قابل پیمایش باشد را می تواند با این روش ها پیمایش کرد مثل: string و ... .

جلسه هشتم – آشنایی با Destructuring آرایه ها و آبجکت ها

let [item1, item2] = list

کد فوق عضو اول و دوم از list را به ترتیب به آیتم های داخل لیست پاس می دهد به این ویژگی array destructuring میگویند.

let a,b

[a, b] = [1, 2] 🡺 هردو متغیر را مقدار دهی می کنیم

مقدار دهی پیش فرض در array destructurnig

let [a, b=1] = [2]

در کد فوق a برابر با 2 میشود و b چون مقداری وجود ندارد مقدار پیش فرض خود را می گیرد.

حتی می توان در سمت راست تساوی تابعی قرار داد که خروجی اش آرایه است.

let [item1, , , item2, item3] = list 🡺 عضو دوم و سوم آرایه نادیده گرفته میشوند

let [a, b, ..c] = list

در کد فوق عضو اول و دوم در a و b ریخته می شوند و مابقی عضو های دیگر در c ریخته می شوند اما …c باید آخرین باشد.

Object destructuring

let obj = {

name: ‘hesam’,

age: 26,

foo: () => {

console.log(‘roocket’)

}

}

Let { name, age, foo } = obj

عیناً نظیر به نظیر پراپرتی و متد های obj در شی جدید ریخته می شوند و بر خلاف array destructuring در اینجا ترتیب مهم نیست و براساس کلید اعضا گرفته می شوند.

برای تغییر نام یک پرارپرتی یا متد در شی جدید بصورت زیر عمل می کنیم:

let { name, age, foo:bar } = obj

در کد فوق متد foo از obj با identifier جدید bar در شی جدید قرار می گیرد.

همانند آرایه ها در اینجا نیز می شود مقدار پیش فرض تعیین کرد.

این مباحث بسیار مهم هستند و خیلی کاربردی اند.

جلسه نهم – آشنایی با Template String

در جاوا اسکریپت ES5 برای رفرنس دادن به متغییر در وسط یک رشته دردسر زیادی داریم و هی باید “ یا ‘ رو ببندیم و از + استفاده کنیم برای چسباندن مقدار داخل متغیر به رشته اما در ES6 بحث Template String رو داریم که اینکار رو بسیار آسان کرده است.

let text = ` hesam

mosavi`

حتی فاصله ها در نوشته نیز حفظ می شوند.

`hello ${1 + 1}` 🡺 حاصل 1 + 1 در متن قرار می گیرد

`hello ${obj.name}` 🡺 پراپرتی شی در متن قرار می گیرد

اگر جلوی ${ } از \ استفاده کنیم دیگر عین عبارت نوشته می شود و به متغیر یا محاسبات رفرنس داده نمی شود.

جلسه دهم – کار با کلاس ها

برنامه نویسی شئ گرایی از دنیا واقعی الهام گرفته شده است.

کلاس حکم نقشه ساخت را دارد برای شئ هایی که از روی آن ساخته می شوند. در پشت پرده جاوا اسکریپت هنوز از توابع سازنده یا constructor function استفاده می کند ولی در ES6 کلاس ها رو تعریف کرده اند.

Class car {

constructor(brand, model, year) {

this.brand = brand

this.model = model

this.year = year

}

}

در تعریف کلاس به جای استفاده از کلمه کلیدی function از class استفاده می کنیم مانند کد بالا. دسترسی به proto شی همانند قبل است هیچ تفاوتی با حالت استفاده از تابع سازنده ندارد.

جلسه یازدهم – ارث بری در کلاس ها

داستان ارث بری در کلاس ها ساده تر شده است؛

Class student extends human {

Constructor() {

Super()

}

}

در ارث بری پدر نیز اجرا می شود و باید از super([arguments]) در فرزند استفاده کنیم تا به پراپرتی های کلاس پدر دسترسی داشته باشیم همانند this. پارامتر های این متد، پارامتر های پدر کلاس می باشند. اگر متدی از پدر صدا زده شود به صورت، super.functionOnParent([arguments]) است. در اصل super به constructor کلاس پدر اشاره می کند. متد ها و پراپرتی ها را می شود override کرد.

جلسه دوازدهم – متدهای Static و توسعه کلاس های داخلی

هر متدی را که static کنیم بدون ساخت instance از کلاس می تواند از متد استفاده کرد.

static run() { }

متدی که استاتیک باشد در هنگام ایجاد نمونه دیگر در دسترس نیست و متد پدر را به ارث می برد. متد فرزندی اگر پدر را صدا بزند و خودش استاتیک باشد متد پدر نیز باید استاتیک باشد.

توسعه دادن شئ های موجود در جاوا اسکریپت:

class AdvanceArray extends Array {

find (value) {

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

return val.length === 0 ? ‘Error’ : val[0]

}

}

let list = new AdvanceArray(1,2,3..)

جلسه سیزدهم - کار با Getter و Setter در کلاس ها

در جاوا اسکریپت برای پراپرتی ها دسترسی های private، protected اینا تعریف نشده است.

class Foo {

constructor () {

This.log = []

}

set current(value)

This.log.push(value)

}

}

let bar = new Foo()

bar.current = ‘A’

در کد فوق به کمک set ایی که تعریف کردیم می توانیم متد را همانند پراپرتی برای افزودن داده استفاده کرد.

get latest() {

if (this.log.length === 0) {

return undefined

}

Return this.log[this.log.length – 1]

}

متد فوق که getter است به صورت پراپرتی صدا زده می شود و () ندارد در هنگام فراخوانی. در زبان های دیگر این getter پراپرتی های کلاس را قابل دسترس می کند اما در جاوا اسکریپت این ویژگی را ندارد چراکه همه پراپرتی ها عمومی یا public هستند.

جلسه چهاردهم – ویژگی های جدید Object ها

اگر بخواهیم یک پراپرتی از آبجکتی را برابر با مقدار متغیری کنیم اگر نام پراپرتی و متغیر یکی باشند در آبجکت فقط نوشتن key کافی است و لزومی ندارد متغیر به عنوان value را در مقابل اش بنویسیم.

let name = “hesam”

let obj = {

name,

[keyname]: “some thing here”,

sayHello () {

}

}

همچمین در کد فوق [keyname] داینامیک است و keyname نام متغیری است که قرار است مقدارش به عنوان نام برای پراپرتی قرار گیرد.

در ES6 به جای قرار دادن متد در مقابل یک key میتواند نام متد را نوشت و () در جلوی آن بگذاریم و مستقیم آن را تعریف کنیم.

در نام متد اگر فاصله میان کلمات وجود داشت مانند “aran abdi”() در هنگام فراخوانی باید در داخل [“aran abdi”]() صدا زده شود: obj.[“aran abdi”]()

جلسه پانزدهم – آشنایی و کار با Symbol ها

Symbol یک نوع داده است (data type). این داده یک مقدار unique یا یکتا در خود دارد که قابل نمایش نیست اما در طول برنامه خود جاوا اسکریپت آن را مدیریت می کند و در آن یک مقدار واحد قرار می دهد.

let symbol1 = Symbol()

برای ایجاد نمونه از Symbol کلمه کلیدی new قبل از Symbol دیگر نوشته نمی شود. در کد فوق symbol1 حاوی یک مقدار یکتا است که نمی دانیم چیست و قابل مشاهده نیست.

اصلا کجا استفاده می شود؟

می توان از آن به عنوان یک هویت استفاده کرد برای شئ ها مثلا:

let obj = {

name: “aran”,

[symbol]: 22

}

برای اینکه بخواهیم بدانیم یک Symbol برای چیست و توضیحاتی داشته باشد بصورت زیر توضیحات را به Symbol پاس میدیم اما این مورد کاری نمی کند و تاثیری ندارد، فقط برای خوانا تر شدن کد است.

Symbol(‘descriptions go here’)

Symbol در forEach و حلقه ها قابل استفاده نیست.

let Symbol.for(‘fullname’)

let SymbolN = Symbol.for(‘fullname’)

در کد فوق مقداری که در خط اول تعریف شده است در متغیر SymbolN ریخته می شود.

اگر به کمک متد ساخته شود Symbol.for(‘fullname’) در طول برنامه ثابت است اما Symbol (‘fullname’) هر بار که فراخوان شود حاوی یک مقدار جدید است.

Symbol.replace به کلیدی که از قبل در جاوا اسکریپت تعریف شده است دسترسی می دهد.

class ReplaceX {  
 constructor (value) {

this.value = value

}

[Symbol.replace](string) {

Return ‘hesam’

}

}

console.log(‘foo1’.replace(new ReplaceX(‘bar’)))

در کد فوق مقدار را با hasam جایگزین می کند.

جلسه شانزدهم – متد های جدید در آرایه ها

در ES6 دو تا متد برای ایجاد کردن آرایه ها اضافه شده است.

let list1 = [1,2,3,4]

در ES5 برای ساخت آرایه از کد فوق استفاده می کردیم اما حالا کد زیر را نیز می توان استفاده کرد برای ایجاد یک آرایه

let list2 = Array(1,2,3,4)

در کد فوق اگر درون () چیزی نوشته نوشت صرفاً یک آرایه میسازد اگر فقط یک عدد پاس داده شود یک آرایه میساز که به تعداد آن عدد عضو خالی دارد. نوشتن new لازم نیست.

let list = Array.of(1,2,3,4)

.of متدی استاتیک ایی است که اگر فقط یک عدد به Array پاس داده شود درون آرایه ریخته میشود و مثل حالت قبل نیست که به تعداد مقدار وروی آرایه n عضوی بسازد.

let list2 = Array.from(list1)

.from یک متد استاتیک دیگر است که آرایه list2 را بر اساس list1 میسازد.

Array.from(list, item => item \* 5)

کد فوق همانند .map بر روی آرایه ها عمل می کند.

در ES6 متد .find رو داریم:

list.find(item => item == 20)

find همانند .filter عمل می کند با این تفاوت که .filter تمامی مواردی که صدق می کنند را بر می گرداند اما .find به محض پیدا کردن اولین مقدار آن را بر می گرداند و دیگر جستجو متوقف میشود

list.entries()

در جلسه بعد که iterator آموزش داده میشود این نیز آموزش داده میشود، یک iterator برمی گرداند.

let arr = list.entries()

arr.next()

کد فوق یک عضو از آرایه را بر می گرداند و اگر دوباره صدا زده شود عضو بعدی را میاورد. به آخرین که برسد با صدا زدن مجدد مقدار done را بر می گرداند.

for (let item of list.entries()) {

console.log(item)

}

در کد فوق key و value را خودکار می آورد. این متد کمک می کند بدون حلقه ها قابلیت پیمایش را داشته باشیم با صدا زدن متد .next() روی آرایه.

list.fill(value)

مقدار همه اعضا را برابر با ورودی متد قرار میدهد. این متد به عنوان آرگومان دوم پوزیشن شروع را می پذیرد یعنی اگر 2 را به عنوان دومین آرگومان پاس بدهیم از ایندکس 2 این کار را انجام میدهد.

و نهایتاً به اگر سومین ارگومان را نیز به آن پاس دهیم تا آن ایندکس عملیات fill ادامه خواهد داشت. ولی خود مقدار دریافتی را حساب نمی کند تا یک ایندکس قبل تر مقدار عوض میشود.

list.copyWhitin(arg1, arg2)

در کد از ایندکس arg2 به بعد هرچه که باشد را به پوزیشن ایندکس arg1 کپی می کند.

اما توجه داشته باشید که عضو های که کپی شده اند همچنان در جای خود می مانند.

اگر آرگومان سوم پاس داده شود ایندکس پایان کپی کردن را تعیین می کند. یعنی تا یکی قبل از این آرگومان سوم کپی مش شود برای کپی فقط یک عضو ایندکس ها را ایندکس های پشت سر هم لحاظ می کنیم مثلا:

list.copyWhitin(0, 3, 4)

در کد فوق مقدار داخل ایندکس 3 را در ایندکس 0 میریزد فقط.

جلسه هفدهم – متد های جدید در اعداد

Number.isNaN(num) چک می کند که آیا NaN هست یا خیر. .isNaN یک متد سراسری است.

Number.isFinit(num) بررسی می کند که آیا عدد محدود است و نامحدود نباشد.

Number.isInterger(num) بررسی می کند که آیا عدد صحیح است یا خیر.

Math.sign(num) اگر عدد مثبت باشد 1 را بر می گرداند اگر عدد 0 باشد همان 0 را بر می گرداند و اگر عدد منفی باشد 1- بر می گرداند. و اگر NaN باشد همان NaN را بر می گرداند.

Math.trunc(num) همانند floor اعداد را گرد به پایین می کند و تفاوت آن ها در برخورد آن ها با اعداد متفی است، در floor عدد منفی به سمت کمتر می رود اما در trunc عدد منفی به سمت + می رود یعنی -2.99 را floor به 3- تبدیل می کند اما trunc آن را به -2 تبدیل می کند.

جلسه هجدهم – متدهای جدید در رشته ها و آبجکت ها

متدهای جدید string ها

text.includes() این متد در text به بررسی می کند که آیا ورودی ای دریافت شده وجود دارد. به حروف بزرگ و کوچک حساس است، و پارامتر دوم ایندکس شروع کردن را می گیرد و از آن ایندکس بعد جستجو می کند فقط، خروجی این متد true یا false است اما indexOf ایندکس آن حرف یا حرف اول آن عبارت را بر می گرداند و اگر وجود نداشت 1- را بر می گرداند.

text.startWith() بررسی می کند که آیا رشته با این مقدار که در ورودی دریافت شده آغاز می شود یا خیر، پارامتر دوم ایندکس شروع کردن را می گیرد.

.endsWith() بررسی می کند که آیا رشته با آن مقدار دیافتی پایان می یابد یا خیر. پارامتر دوم به جای اینکه آخرین ایندکس را در نظر بگیرد این پارامتر را به عنوان انتهای مد نظر در نظر می گیرد.

متدهای جدید object ها

let obj3 = Object.assign(obj1, obj2)

متد فوق دو تا آبجکت ورودی را باهم ادغام می کند و حتی اگر نتیجه در obj3 هم ریخته نمی شد مستقیم نیتجه در ابجکتی که به عنوان اولین ورودی دریافت شده است ریخته می شود.

اگر پراپرتی مشترک داشته باشند آبجکت ها موارد آبجکتی که دومین ورودی متد است بر روی پراپرتی آبجکت اول بازنویسی می شود و پراپرتی آبجکت اول حذف می شود.

اگر آبجکت ها از کلاس ساخته شوند به سادگی ترکیب می شوند اما کلاس سازنده آبجکت اولی همچنان کلاس سازنده اصلی است. حتی کلاس سازنده آبجکتی که نتیجه assign را درونش میریزیم همان کلاس سازنده آبجکت اول است.

اگر نخواهیم آبجکت اول تغییر کند و فقط آبجکتی جدیدی ساخته شود که ادغام هر دو آبجکت است بصورت زیر عمل می کنیم.

let obj3 = Object.assign({}, obj1, obj2)

در حالت بالا constructor دیگر Object است همچمین هر تعداد آبجکت که به متد ذکر شده پاس دهیم مشکلی نست و همگی ادغام می شوند و نتیجه در اولین آبجکت پاس داده شده ریخته می شود.

Object.setPrototypeof(obj, prototype)

متد فوق یک پروتوتایپ برای آبجکت پاس داده شده قرار می دهد که پروتوتایپ پاس داده شده می تواند شئ، کلاس یا null باشد.

جلسه نوزدهم – آشنایی و کار با Iterator ها

در آرایه ها Symbol.iterator وجود دارد برای این است که امکان پیمایش شدن را دارند.

اگر بخواهیم آرایه ای قابل پیمایش نباشد دیگر کافی است کد زیر را بنویسیم.

arr[Symbol.iterator] = null

اگر بخواهیم آرایه ای را بدون استفاده از حلقه ها پیمایش کنیم بصورت زیر عمل می کنیم.

let arr = [2,4,6]

let arrIterable = arr[Symbol.iterator]()

console.log(arrIterable.next())

متد .next() بار اول مقدار داخل ایندکس 0 را بر می گرداند و هر بار که صدا زده شود مقدار داخل ایندکس بعدی را برمی گرداند و به انتهای آرایه که برسد اگر مجدد صدا زده شود done بر می گرداند.

arr[Symbol.iterator] = fuction () {}

تابع فوق next() را بازنویسی می کند. در تابع فوق this به آرایه اشاره می کند و نمی توان arrow function قرار داد چراکه this دیگر در دسترس نخواهد بود. می شود متد اضافی در آن قرار داد.

اضافه کردن قابلیت پیمایش به Object

objName[Symbol.iterator] = function() {

let posts = this.posts

let step = 0

return {

next() {

let obj = {

done: step >= posts.length,

value: posts[step]

}

step++

return obj

}

}

}

حال دیگر شئ توسط For-of و forEach قابل پیمایش است چون Symbol.iterator آن تعریف شده است و پست ها را که یک آرایه در آبجکت است را برمی گرداند.

جلسه بیستم – آشنایی و کار با Generator ها

بین Generator ها و Iterator ها رابطه نزدیکی وجود دارد. می توانیم تابعی داشته باشیم که توقف داشته باشد و این خیلی کاربردی است.

Generator تابعی است با امکان متوقف شدن که برای تعریف کافی است بنوسیم function\* به جای function خالی.

Function\* createNames() {

yield ‘hessam’

console.log..

Yield ‘ali’

console.log..

return ..

}

تابع فوق به هر yield که می رسد متوقف می شود و هر بار یک yield را بر می گرداند.

let namesIterable = createNames()

console.log(namesIterable.next)

در حلقه نیز میشود استفاده کرد از آن

for (const name of createNames()) {..}

برای گرفتن خروجی همه yield ها از console.log([..createNames()]) استفاده می کنیم.

کاربرد در Object ها

ساده سازی کد جلسه قبل برای پیمایش Object

\*[Symbol.iterator]() {

yield\* this.posts

}

متد فوق را در آبجکت می نویسیم حال آبتجکت قابل پیمایش است.

جلسه بیست و یکم – آشنایی با جهنم Callback ها

در جاوا اسکریپت بر خلاف PHP برنامه منتظر خروجی تابع نمی ماند و کار خود را ادامه می دهد و کرد های دیگر را اجرا می کند اما کد قبلی همچنان در پس زمینه تا زمانی که تمام شود ادامه پیدا می کند.

مفهومی داریم به نام عملیات غیر همزمان یا asynchronous operation که callback ها در این مواقع به کار می آیند.

Callback ها مربوط به ES5 هستند.

Function doSomething(callback) {

setTimeout( () => {

callback(‘run z’)

}, 2000)

}

doSomething( function(data) {

console.log(data)

})

در کد فوق به کمک setTimeout روند گرفتن پاسخ از سرور را شبیه سازی کرده ایم که غالباً قدر زمان بر است و یک تابع را به تابع مان پاس داده ایم تا داده ای را که از تابع برگشت داده میشود را نمایش دهد. کد فوق پس از 2 ثانیه اجرا می شود و خروجی میدهد.

وبسایت callbackhell.com به مشکلات callback ها پرداخته است.

اگر callback های تو در تو داشته باشیم که پیچیده میشود و مدیریت آن خیلی سخت می شود.

نکته هرجا که بخواهیم تابع بدون خروجی متوقف شود از return تنها استفاده می کنیم.

مثالی که در ویدیو حسام موسوی می زند آنقدر پیچیده است و این مشکل توسط promise حل میشود البته در ES8 حالت جدید تری نیز اضافه شده که حتی کار را از promise نیز آسانتر می کند.

جلسه بیست و دوم – آشنایی و کار با promise ها

پرامیس ها همانند کال بک ها پروسه های زمان بر را برای ما مدیریت می کنند. این روش سه تا مدت جدید دارد: catch، then، finally.

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

setTimeout(() => {

resolve(..)

}, 2000)

})

Resolve خروجی را به ما پاس می دهد اما reject را می توان در جای نوشت که می خواهیم خطا را داشته باشیم.

firstPromise

.then(function(data) {

console.log(data)

})

.catch(err => console.log(err))

در کد فوق اگر resolve باشد در then می توان خروجی را گرفت اما اگر reject باشد در catch ارور را دریافت می کنیم.

برای دریافت درخواست Ajax از این حالت استفاده می کنیم. زنجیره پرامیس ها کمک می کند که دوباره نیاز نباشد .then قرار بگیرد.

جلسه بیست و سوم – آشنایی و کار با promise ها – بخش دوم

استفاده به صورت static

function doSomething() {

let name = ‘aran abdi’

return Promise.resovle(name)

یا

return Promise.reject(‘Error!’)

}

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

اجرای مجموع Promise ها:

Promise.all(iterable list of promises)

در صورتی .then اجرا میشود که همه promise ها resolve بشوند.

Promise.race(iterable list of promises)

.race همانند .all کار می کند با این تفاوت که هرکدام که زود تر نتیجه داشته باشد آن را بر می گرداند حال چه resolve شود و چه reject شود.

جلسه بیست و چهارم – آشنایی با Maps

تا قبل از ES6 فقط از آرایه ها و لیست برای داشتن کالکشنی از داده ها استفاده می کردیم و اگر می خواستیم داده ها ایی به صورت key و value داشته باشیم از آبجکت ها باید استفاده می کردیم.

اما در ES6 دو تا ماژول اضافه شده به نام Maps و Sets که در جلسه آینده به Sets می پردازیم.

ساخت یک کالکشن key و value ایی

let list = new Map()

افزودن عضو به کالکشن ساخته شده

list.set(key, value)

key می تواند هرچیزی باشد حتی Symbol پس key ها نیز با این روش می توانند داینامیک باشند.

در Map نمی شود به صورت [ ] مقادیر یک key را دریافت کرد بلکه باید از متد .get استفاده کنیم:

list.get(key)

می توان در یک متغیر آبجکت و فانکشن خالی قرار داد و به عنوان کلید آن ها را پاس داد اما مستقیم نمی توان آن ها را نوشت.

.size تعداد اعضا را برمی گرداند و جایگزین .length است و باید از این متد استفاده شود.

.delete(key) عضو را حذف می کند و .clear() کل اعضای کالکشن را حذف می کند و آنرا خالی می کند.

این کالکشن بر خلاف آبجکت ها که راه جایگزین بودند در ES5 قابل پیمایش هستند.

list.forEach( (value, key) => {

console.log(key, value)

})

.has() مقداری را به عنوان ورودی می گیرد و در میان key ها جستجو می کند و بررسی می کند که آیا وجود دارد یا خیر.

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

console.log(key, value)

}

نکته: Symbol در .maps نشان داده می شود.

.entreis() این متد key و value ها را بر می گردند.

.keys() فقط key ها را بر می گرداند.

.values() فقط value ها را بر می گرداند.

اگر آرایه ای را به عنوان ورودی به new Map() پاس دهیم یک کالکشن key و value ایی از آن آرایه میسازد:

let list = new Map(arr)

جلسه بیست و چهارم – آشنایی و کار با Sets

این کلاس یک کالکشن میسازد که دیتای آن یکتا هستند (unique) و اگر سعی در وارد کردن داده ی تکراری داشته باشیم اصلاً وارد نمی شود.

let list = new Set()

برای افزودن عضو به جای .set متد .add رو داریم: list.add(item)

console.log(list) اعضا را بر می گرداند و .get() نداریم چراکه اعضا تکی گرفته نمی شوند و key و value نیست. اما .delete() و .clear() را داریم همچنان ولی اینجا .delete() بر اساس value حذف می کند، برخلاف Map، چون در اینجا key معنی ندارد دیگر.

.entries() این متد value ها را می گیرد. .keys() و .values() هردو یک چیز را بر می گردانند و تفاوتی ندارند.

.has() بررسی می کند که آیا مقدار دریافت شده در میان اعضا وجود دارد یا خیر.

console.log([..list]) این کد کالکشن ساخته شده با set را به عنوان آرایه برمی گرداند.

همچنین می توان آرایه ای را به آن پاس داد و از روی آن کالکشن را ساخت اگر در این شرایط آرایه اعضای تکراری داشته باشد فقط یکی در نظر گرفته میشوند و اعضای تکراری نادیده گرفته میشوند.

let list = new Set(arr)

جلسه بیست و ششم – آشنایی با weakMaps و weakSets

این دو به نسبت Maps و Sets متد های کمتری دارند اما این تنها تفاوت نیست و نمی شود گفت این دو ضعیف تر هستند و از این حرف ها.

let list = new WeakMap()

در WeakMap نمی توان key ها را داینامیک تعریف کنیم یا حتی Symbol قرار داد، تنها می توان آبجکت را به عنوان key قرار داد. ضمناً قابل پیمایش نیست و نمی تواند آن را در حلقه استفاده کرد.

افزودن عضو به کالکشن WeakMap

list.set({ali: ‘iran’}, ‘ali’)

اما این روش آنچنان کاربردی نیست و فقط برای بهبود معماری به کار می آید و خیلی کم پیش می آید از آن استفاده شود مگر آنطور که گفتیم بخواهیم معماری برنامه رو بهتر کنیم.

در جاوا اسکریپت window یک آبجکت سراسری (global) است. window.car را میشود به صورت Map یا weakMap ساخت اما اگر عضوی که با این key ساخته شده است را هنوز باقی می ماند.

Delete window.car

اما در weakMap خبری از دیتا نیست دیگر و این یکی از مزایای خوب هست.

weakSet نیز همانند WeakMap است:

در weakSet تنها می توانیم آبجکت قرار دهیم:

New WeakSet([{a: 1}, {b: 2}])

.has نیز false برمی گردد اگر delete انجام دهیم.

جلسه بیست و هفتم – کار با Reflect API – بخش اول

Reflect Api و proxy Api رابطه نزدیکی دارند.

ابتدا باید بدانیم Meta Programming چیست؟

یک تکنیک برنامه نویسی هست که هر زبانی به روش خودش آن را انجام می دهد. برای مدیریت روی کدهای خود برنامه است برای مثال typeof از این موارد است.

Reflect نه شئ است و نه کلمه کلیدی است در جاوا اسکریپت و بدیهی است که نمی شود آنرا new کرد.

یکسری متد در اختیار ما قرار می گیرد که برای مدیریت و ... کد ها است.

متدهای Reflect و با حالت های دیگری که صدا می زنیم تقریبا یکی اند اما مقداری متفاوت هستند. حالت های دیگر یعنی روی شئ هایی این متد ها ممکن است وجود داشته باشند.

ترکیب Reflect و Proxy ما را قادر میسازد کارهای خیلی بیشتری انجام دهیم.

let obj = Reflect.construct(car, [..])

متد فوق برای ساخت object به کار می رود از روی یک کلاس یا متد سازنده که آنرا به عنوان وروری اول به این متد پاس می دهیم و پارامتر ها را در قالب یک آرایه به عنوان ورودی به این متد پاس می دهیم.

پروتوتایپ در این روش با روش های معمولی ساختن شئ شرایط یکسانی دارد البته می توان به عنوان ورودی سوم یک پروتوتایپ پاس دهیم تا پروتوتایپ شئ ای که ساخته می شود بر آن اساس باشد. این ورودی سوم می تواند کلاس یا متد سازنده باشد.

روش فوق در ایجاد کتابخانه ها و فریم ورک ها کاربردی است چراکه به ما داینامیکی و انعطاف میدهد اما درحالت عادی و ساخت برنامه های عادی زیاد کاربر ندارد.

Reflect.apply(funName, obj, [args])

کد فوق مثل apply عمل می کند اما داینامیک است.

conslole.log(Reflect.getPrototypeof(obj))

کد فوق پروتوتایپ یک شئ را بر می گرداند.

conslole.log(Reflect.setPrototypeof(obj1, obj2))

کد فوق پروتوتایپ obj2 را برای obj1 قرار می دهد.

جلسه بیست و هشتم – کار با Reflect Api – بخش دوم

در این روش ما انعطاف و داینامیک بودن را داریم و مزیت خوبی است برای شرایطی که به آن نیاز داریم.

Reflect.defineProperty(obj, ‘key’, {value: ..})

کد فوق امکان افزودن یک پراپرتی به آبجکتی را به ما میدهد.

در هنگام تعریف پارامتر سوم می توان option های دیگری را تعریف کرد:

{

value: ..,

writable: false

}

کد فوق دسترسی ویرایش کردن را برای مقدار تعریف شده مسدود می کند.

Reflect.deleteProperty(obj, ‘key’)

دستور بالا پراپرتی پاس داده شده را از آبجکت حذف می کند.

delete obj.name 🡺 یک ویژگی یا اتربیوت را از شئ حذف می کند

console.log(Reflect.get(obj or arr, ‘key’) 🡺 مقدار کلید پاس داده شده را بر می گرداند

Reflect.has(obj, value) 🡺 بررسی می کند که آرگومان پاس داده شده در کلیدها هست

Reflect.set(obj, ‘key’, value) 🡺 یک پراپرتی اضافه می کند و یا آپدیت می کند

Reflect.ownKeys(obj) 🡺 همه ی کلید های آرایه را در قالب آرایه بر می گرداند

Reflect.isExtensible(obj) 🡺 true و false 🡺 توسعه پذیری را بررسی می کند

Reflect.preventExtensions(obj) 🡺 از قابل توسعه بودن شئ جلوگیری می کند

اگر حالت بالا را برای یک شئ قرار دهیم دیگر متد و پراپرتی و اینا کم و زیاد نمی شوند و تغییر نمی کنند.

جلسه بیست و نهم – کار با Proxy Api

پراکسی در دنیای اینترنت یک تونل است برای ارتباط در اینترنت از طریق یک کانال.

در جاوا اسکریپت اگر proxy وجود نداشته باشد اطلاعات شئ را بگیریم در برنامه مستقیم شئ به برنامه پاسخ می دهد اما با وجود proxy این وظیفه دیگر بر عهده proxy است و نقش رابط را دارد.

let handler = {

get(obj, key) {

return ..

}

}

let proxy = new Proxy(obj, hander)

proxy.property

کد فوق به ما این امکان را می دهد که به پراپرتی یک شئ به طریقی که می خواهیم دسترسی پیدا کنیم و آن را از شئ proxy که یک نمونه از کلاس Proxy است در دسترس داریم و صدا می زنیم.

متد هایی که وجود دارند را در handler بازنویسی می کنیم.

has(obj, key) {

return Reflect.has(obj, key)

}

console.log(Reflect.has(proxy, ‘year’)

متد has را همانگونه که در بالا گفتیم درون handler می نویسیم، کد فوق متد has را بازنویسی می کند با ترکیب پراکسی و رفلکت.

اما صدا زدن پراپرتی و متد روی نمونه ای که از Proxy ساخته ایم جالب نیست بنابر این:

Reflect.setPrototypeof(obj, proxyObj)

کد فوق پروتوتایپ شئ پراکسی را برای شئ ای که روش کار می کنیم قرار می دهد و اینگونه پراپرتی و متد را روی خود شئ صدا می زنیم با این تفاوت که پراکسی آن را به ما میدهد.

جلسه سی ام – کار با Proxy Api – بخش دوم

بخش apply پراکسی ها

Apply را در handler قرار می دهیم سپس proxy() را مثه متد (تابع) فراخوانی می کنیم، proxy را خودمان به کمک new proxy ساخته ایم.

Target را خودش به آن پاس میدهد فقط آرگومان ها و آرگومان های برگشتی پاس داده می شوند، proxy() متد apply را اجرا می کند. و apply مثه متد constroctur است یه جورایی.

اگر بخواهیم یک پراکسی قابلیت لغو شدن را داشته باشد در هنگام تعریف به جای new Proxy می نویسیم، Proxy.revocable() و دیگر new را نمی نویسیم.

Let {proxy, revoke} proxy.revocable({}, handler)

میشود هردو را در یکی ریخت اما کار فوق آن ها را تفکیک می کند. حال هرکجا revoke() را فراخوانی کنیم پراکسی لغو می شود.

اگر پراکسی را به صورت prototype به شئ بدهیم دیگر آبجکت {} (target) proxy است.

متد های دیگری نیز برای تعریف شدن وجود دارند که همانند متد های شرح داده شده هستند.

جلسه سی و یکم – نصب و راهندازی webpack

اگر ماژولار کردن و پکیج بندی کد ها نباشد باید کد ها در یک فایل نوشته شوند و مدیریت کد ها با بزرگتر شدن پروژه بسیار سخت می شود. ماژولار کردن یعنی کد ها را بسته بندی کنیم و هرجا که نیاز بود از آن استفاده کنیم.

پکیج هایی برای ماژولار کردن آمدند که سنگین بودند و مشکلاتی داشتند بنابر این محبوب نشدند.

توجه داشته باشید که فایل بندی به تنهایی فقط قطعه قطعه کردن کد ها است و ماژولار کردن نیست.

Node.js سیستم ماژولار کردن خود را دارد. اما در ES6 نوع دیگری از ماژولار کردن را داریم.

import و export را داریم.

اما این روش مستقیم قابل اجرا نیست و باید از ابزاری استفاده کنیم به نام webpack.

با build کردن پروژه در نهایت کل فایل ها کامپایل می شوند و در یک فایل قرار می گیرند.

در گذشته gulp وجود داشت اما امروزه دیگر از webpack غالباً استفاده می شود.

یادگیری webpack خودش یک دور دیگر می طلبد که در راکت وجود دارد.

برای نصب webpack از npm استفاده می کنیم که خود npm را باید به همراه node.js نصب کنیم.

دستور npm init را برای افرودن npm به پروژه وارد می کنیم و گزینه ها را پاسخ میدهیم.

با اینکار فایل package.json ساخته می شود.

npm install webpack webpack-cli

دستور فوق وب پک را برای ما نصب می کند. فایل webpack.config.js را باید بسازیم در پروژه و تنظیماتی را وارد کنیم برای شناساندن پروژه به وب پک.

وابستگی ها در دایرکتوری node\_modules قرار می گیرند.

در فایل package.json درون کلید script کد زیر را اضافه می کنیم.

“script”: {

“build”: “webpack-cli –mode=production”

}

حال npm run build را در ترمینال میزنیم تا پروژه build شده ایجاد شود.

برای فراخوانی یک ماژول در فایلی که به ماژول نیاز دارد از import استفاده می کنیم:

import key from “file address”

هر بار که کدی را تغییر می دهیم باید یکبار پروژه را build کنیم که کار تاقت فرسایی است اما کد زیر را نیز به داخل script همانند حالت build اضافه می کنیم:

“watch”: “webpack-cli –mode=development –watch”

حال با زدن npm run watch دیگر هر تغییری اعمال کنیم خودکار build گرفته می شود و نیتیجه را سریع تر می توان مشاهده کرد.

در سند مربوط به ماژول برای در دسترس بودن آن در فایلی که به آن احتیاج دارد مواردی که باید در دسترس باشند را باید export کنیم:

let value = {..}

export default value

حال اگر نام متغیر فوق را در هنگام import وارد کنیم دیگر در فایل دیگر در دسترس است.

<script src=”./build/bundle.js”></script>

به جای سندی که قبلا در index.html فراخوانی می کردیم باید کد فوق قرار گیرد.

در هنگام انتقال پروژه دایرکتوری node\_modules ارسال نمی گردد و در پروژه دریافت شده باید npm install را بزنیم تا این وابستگی ها مجددا نصب شوند.

جلسه سی و دوم – آشنایی و کار با ماژول ها

export گرفتن در جاوا اسکریپت روش های مختلفی دارد.

export {

test,

name,

..

}

کد فوق مقادیر را همگی اکسپورت می کند.

مقدار کلید های که در بالا قرار گرفته اند در هنگام import بصورت زیر دریافت می شوند.

Import {test, name,..} from ‘address’

فرقی ندارد چه متد و چه پراپرتی یا به عبارتی متغیر ها و توابع را می توان همگی اکسپورت گرفت حتی کلاس ها را میشود اکسپورت کرد.

در آدرس دهی فایل ها هنگام ایمپورت نیازی به نوشتن پسوند .js نیست.

می شود ماژولی را در ماژول دیگری نیز فراخوانی کرد و این فقط محدود به فایل js اصلی نیست.

console.log(`%c ${message}`, ‘color:green’)

در کد فوق رنگ متنی که در console نمایش داده شود را تغییر داده ایم به سبز

export default در سند فقط یکبار می تواند تعریف شود و نام کلاس یا ... را در مقابل اش قرار میدهیم اما در فایل همچنان می تواند export های معمولی دیگری را نوشت.

برای فراخوانی export default دیگر به {} نیازی نیست و می توان هر اسمی را نوشت و خودکار کلاس یا ... به آن داده می شود.

اگر در هنگام import بخواهیم نام موردی که دریافت میشود را تغییر دهیم بصورت زیر عمل می کنیم.

{ err as error }

حل متغیر، متد یا هرچیزی دیگری که در ماژول err است در این فایل که ماژول را فراخوانی کرده ایم error است و با این نام به آن دسترسی داریم. در array destructuring این به جای as علامت : را قرار می دادیم که در اینجا نمیشود آن را استفاده کرد.

Import \* as .. from “..” 🡺 همه مواردی که اکسپورت شده اند را می گیرد

Key.default ??????????????????????????????????????????

این مباحث در فریم ورک ها خیلی استفاده میشوند و به شدت دیده می شوند. همچنین برای استفاده از پکیج هایی که دیگران نوشته اند نیز کاربرد دارد.

پکیج underscore برای کار کردن با آرایه ها و آبجکت ها و ... است و امکانات خیلی بیشتری به ما میدهد.

برای نصب آن دستور npm i underscore -D را در ترمینال وارد می کنیم.

برای فراخوانی آن از کد زیر استفاده می کنیم.

Import \_ from ‘underscore’

در کد فوق به جای \_ میتوان نام دیگری قرار داد اما در مستندات خودش اینگونه نوشته شده است.

مثلا \_.first(arr) عضو اول آرایه را بر می گرداند.

جلسه سی و سوم – کامپایل جاوا اسکریپت با Babel

Babel برای پشتیبانی کردن پروژه از مرورگر های قدیمی است و ابزار بابل رو باید نصب کنیم و کانفیگش می کنیم و خودش دیگر خودکار در هنگام build گرفتن کدها را کامپایل می کند به کدهای قابل درک برای انواع مرورگرهای تعیین شده.

نکته در هنگام استفاده از npm پیام های warn مشکلی ایجاد نمی کنند و روی آن ها حساس نباشید.