Introduction and What Is ES6?

- ES6 هو اختصار ECMAScript 6 و ECMA هي منظمة بتعمل standardization (وضع معايير) للمعلومات على الانترنت و شركة Netscape طلبت من منظمة ECMA انها تعمل standardization للغة الـJavaScript و نتج عن كدا ان منظمة ECMA عملت standardization تحت اسم ECMAScript و كل فترة المنظمة بتنتج إصدار جديد فيه تحديثات و تسهيلات توفر وقت و مجهود و تحل مشاكل و الاصدارES6 نزل سنة 2015 و لذلك البعض بيقول ECMAScript 2015 و لكن الاسم الشائع هو ECMAScript 6

- Node.js و React.js و vue.js و Angular.js بيعتمدوا على الاصدار ES6

- معظم المتصفحات الحديثة بتدعم الـES6 لكن بعض المتصفحات القديمة مش بتدعمه لذلك بنلجأ لاستخدام transpilers لتحويل اكواد الـES6 لاكواد ES5 بحيث ان الاكواد تكون متوافقة مع كل المتصفحات و كلمة transpiler هي كلمة مركبة من translate و compile .. الـtranspilers فيه منها online زي موقع babel و فيه منها offline زي package اسمها babel في gulp.js و اضافات في الـIDE اللي انت بتستخدمه

New Way to Create Variable - Let

- في الـES6 عندنا طريقتيتن نعمل بيهم variable هما (let , const) .. لكن قبل ما نبدأ فيهم محتاجين نعرف شوية معلومات

لو كتبنا كود بالشكل ده

var varName;  // Declaration

console.log(varName)

النتيجة في الـconsole هتكون undefined لأننا عملنا declaration فقط و معملناش assignment للمتغير varName

بينما لو كتبنا كدا

var varName;  // Declaration

varName = 1; // assignment

console.log(varName)

النتيجة هتكون 1

لذلك كنا بنختصر الـdeclaration و الـassignment في سطر واحد بالشكل ده

var varName = 1;

console.log(varName)

- هنبدأ نعمل مقارنة بين الـvar و الـlet و الـconst

- لو عندنا function بالشكل ده

function varTest(){

    var x = 1;  // Declaration and Assignment

    if(true){

        var x = 2; // Re-Declaration and Assignment

        console.log(x)

    }

    console.log(x);

    console.log("Done");

}

varTest();

النتيجة هتكون

2

2

Done

نستنتج من المثال ان الـvar ممكن نعملها Re-Declaration و accessible في كل مكان داخل الـfunction و بالتالي هي function scope .. بينما لو طبقنا نفس المثال باستخدام let بدلا من var بالشكل ده

function letTest(){

    let x = 1;

    if(true){

        let x = 2;

        console.log("Inside If Condition " + x)

    }

    console.log(x);

    console.log("Done");

}

letTest();

النتيجة هتكون

Inside If Condition 2

1

Done

نستنتج من المقارنة هنا ان الـlet مش accessible في كل مكان داخل الـfunction و انما accessible داخل الـ block { } بتاعها فقط و بالتالي هي block scope .. و من ضمن الاختلافات بين الـlet و الـvar انه الـRe-Declaration غير مسموح مع الـlet على عكس الـvar .. مثال

var x = 1;

var x = 2;

console.log(x)

النتيجة 2

بينما لو استخدمنا الـlet بدلا من الـvar بالشكل ده

let x = 1;

let x = 2;

console.log(x)

النتيجة هتكون x has already been declared

- الـvar و الـlet بيقبلوا الـRe-Assignment بمعنى ان المتغير ياخد value جديدة من غير Declaration بالشكل ده

var x = 1;

x = 2;

console.log(x)

النتيجة 2

نجرب مع الـlet

let x = 1;

x = 2;

console.log(x)

النتيجة 2

- من ضمن الاختلافات البسيطة بين الـvar و الـlet هو انك لو حاولت تـaccess متغير بالـvar قبل الـDeclaration النتيجة هتكون undefined بينما لو حاولت تـaccess متغير بالـlet قبل الـDeclaration النتيجة هتكون Reference Error

مثال

console.log(x)

var x = 1;

console.log(y)

let y = 1;

النتيجة

undefined

Uncaught ReferenceError: Cannot access 'y' before initialization

- الـvar بيعمل property باسم المتغير فالـwindow object على عكس الـlet .. مثال

var osama = 1;

let ahmed = 2;

console.log(window.osama)

console.log(window.ahmed)

النتيجة

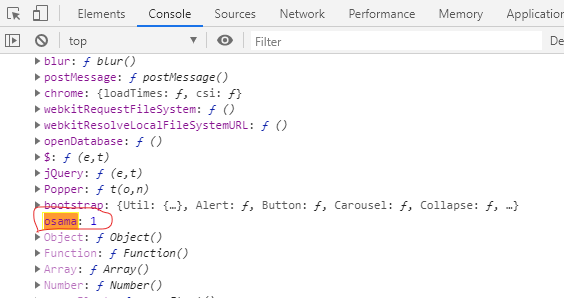
1

Undefined

و لو شوفنا الـwindow object فالـconsole بالشكل ده

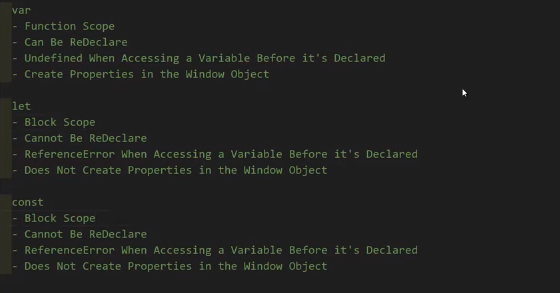
console.log(window)

هنلاقي property باسم osama



و ده كان بيسبب مشكلة اسمها variable scope drama و هي ان المتغير بيكون accessible في الـwindow object و المشكلة دي اتحلت بظهور الـlet و الـconst

- معظم الفروق اللي بتميز الـlet عن الـvar تنطبق على الـconst مع اختلافات بسيطة هنوضحها المحاضرة الجاية و نقدر نلخص الفرق بين الـvar و بين الـlet و الـconst في الصورة دي



New Way to Create Variable - Const

- الـconst هي اختصار لـconstant و معناها الثوابت .. و الثوابت هي عبارة عن متغير بـvalue ثابته و لا يقبل

الـRe-assignment و جرى العرف بين الـdevelopers ان اسم الـconst يكون كله بحروف uppercase

مثال

const SETTINGS = 1;

console.log(SETTINGS)

النتيجة 1

- الـconst زي الـlet لا يقبل الـRe-Declaration .. مثال

const SETTINGS = 1;

const SETTINGS = 2; // or var SETTINGS = 2; or let SETTINGS = 2;

console.log(SETTINGS)

النتيجة

'SETTINGS' has already been declared

- الـconst لازم تعمله Declaration و assignment في خطوة واحدة على عكس الـlet و الـ var

const SETTINGS;

SETTINGS = 1;

console.log(SETTINGS)

النتيجة

Missing initializer in const declaration

- الـconst يعتبر block scope زي الـlet .. مثال

const SETTINGS = 1;

if(true){

    const SETTINGS = 2

    console.log(SETTINGS)

}

console.log(SETTINGS)

النتيجة

2

1

- الـconst ممكن تكون array فاضية و نضيف values فيها لأننا مش بنعمل Re-Assignment للـarray و انما بنضيف عليها مثال

const SETTINGS = [];

SETTINGS.push('HTML', 'CSS', 'JS');

console.log(SETTINGS)

النتيجة

(3) ["HTML", "CSS", "JS"]

بينما لو عايزين نعمل Re-Assignment للـarray هنعمل كدا

const SETTINGS = [];

SETTINGS = ["Osama"]

console.log(SETTINGS)

بما ان الـconstants لا تقبل الـRe-assignment .. النتيجة هتكون

Assignment to constant variable

- نفس الكلام اللي عملناه على الـarray ينطبق على الـobject و هو اننا نقدر نعدل على property فالـobject وانما مانقدرش نعمل Re-Assignment للـobject كله .. مثال

const SETTINGS = {

    name: 'Osama',

    age: 36

};

SETTINGS.name = 'Ahmed'

console.log(SETTINGS)

النتيجة

{name: "Ahmed", age: 36}

بينما لو عملنا Re-Assignment للـobject كله بالشكل ده

const SETTINGS = {

    name: 'Osama',

    age: 36

};

SETTINGS = {

    color: 'red'

}

SETTINGS.name = 'Ahmed'

console.log(SETTINGS)

النتيجة

Assignment to constant variable.

- لو عايز أمنع أي تعديل يحصل على property في الـ object بستخدم method اسمه freeze .. مثال

const SETTINGS = {

    name: 'Osama',

    age: 36

};

Object.freeze(SETTINGS)

SETTINGS.name = 'Sayed'

console.log(SETTINGS)

النتيجة

{name: "Osama", age: 36}

Arrow Function - Syntax and Examples

- الدرس ده هنتكلم عن الـarrow function و امتى نستخدمها بدلا من الـregular function

- قبل ظهور الـarrow function كنا بنعمل الـfunction بالشكل ده

function test(){

    return 2;

}

console.log(test())

النتيجة 2

و من الممكن تخزين الـfunction في متغير بالشكل ده

let test = function(){

    return 2;

}

console.log(test())

النتيجة 2

نفس الـfunction دي نقدر نعملها بطريقة الـarrow function بـsyntax بسيط و سهل .. كل اللي اللي علينا نعمله هو اننا هنحذف كلمة function و نستبدلها بـ => بعد الاقواس بالشكل ده

let test = () => {

    return 2;

}

console.log(test())

النتيجة 2

لو الـbody بتاع الـfunction عبارة عن سطر واحد .. ممكن نلغي الأقواس { } و نلغي كلمة return بالشكل ده

let test = () => 2;

console.log(test())

النتيجة 2

- لو مش عامل parameters في الـ arrow function .. تقدر تلغي الأقواس ( ) و تسبدلها بـunderscore بالشكل ده

let test = \_ => 2;

console.log(test())

النتيجة 2

- نعمل مثال باضافة parameter فالـarrow function

let test = (param) => param \* 2;

console.log(test(16))

النتيجة 32

- فالـarrow function لو عندك parameter واحد .. ممكن تلغي الاقواس ( ) بالشكل ده

let test = param => param \* 2;

console.log(test(16))

النتيجة 32

بينما لو عندك اكتر من parameter يبقى لازم الاقواس تظل موجودة .. مثال

let test = (param1, param2) => param1 \* param2;

console.log(test(10, 30))

النتيجة 300

- الـarrow function مش بديل للـregular function و انما بنستخدمها في أماكن و حالات مناسبة ليها و الحالات الأخرى بنستخدم الـregular function و ده هنتناوله بشكل عملي في باقي الكورس

- الـsyntax الخاص بالـarrow function كدا

// Identifier               => Expression

let test = (param1, param2) => param1 \* param2;

Arrow Function and This Value

- هنبدأ نتناول الفروقات ما بين الـarrow function و الـregular function و أول فرق هو طريقة فهم الـarrow function و الـregular function للـ**this**

الـthis داخل الـregular function بتمثل الـobject اللي نادى عليها .. مثال

هنعمل div في ملف الـHTML واخد id اسمه show .. و هنعمل button واخد id اسمه btn بالشكل ده

<div id="show"></div>

<button id="btn">click</button>

و هنعمل regular function بتغير في محتوى الـshow div بالشكل ده

let test = function(){

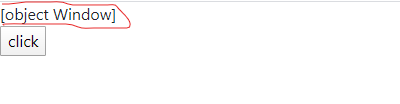
    document.getElementById('show').innerHTML = this

}

// window object called the function

window.onload = test()

الـfunction هنا لما ننادي عليها هتطبع في محتوى الـshow div اسم الـobject اللي نادى عليها بالشكل ده

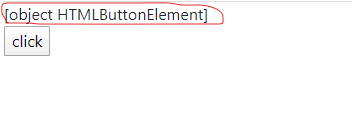


نعمل مثال تاني

// Button object called the function

document.getElementById('btn').addEventListener('click', test)

هنا بمجرد ما نضغط على الـbutton .. الـtest function هتتنفذ و هتطبع اسم الـobject اللي نادى عليها داخل محتوى الـshow div

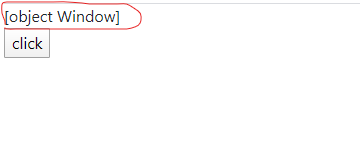


بينما الـthis في الـarrow function بتمثل السياق اللي الـfunction متواجده فيه .. مثال

let test = () => document.getElementById('show').innerHTML = this

document.getElementById('btn').addEventListener('click', test)

لما نضغط على الـbutton النتيجة هتكون



الـarrow function هنا طبعت window لأنها موجودة في الـwindow object

Template Literals

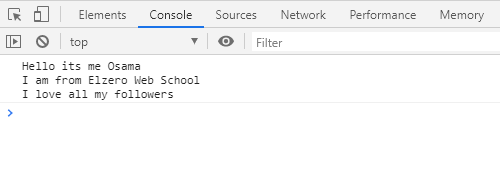
-الـTemplate Literals اوالـTemplate strings من الاضافات الرائعة في الـES6 وهي بتمكنك انك تكتب multiline strings بشكل احترافي اكتر

- قبل اصدار الـES6 .. كنا بنعمل multiline string باستخدام \n عشان نبدأ من سطر جديد .. مثال

let oldWay = "Hello its me Osama\nI am from Elzero Web School\nI love all my followers";

console.log(oldWay)

النتيجة



بعد اصدار الـES6 اصبح كتابة الـmultiline string اسهل و ابسط لأننا مش هنحتاج الـ\n عشان نبدأ سطر جديد و انما كل ما علينا اننا نكتب الـmultiline string بين علامة الـbacktick بدلا الـdouble quotes او الـsingle quotes بالشكل ده

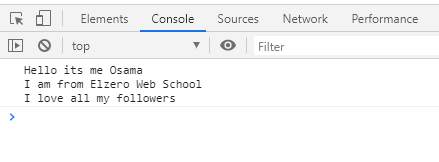
let newWay = `Hello its me Osama

I am from Elzero Web School

I love all my followers`;

console.log(newWay)

النتيجة



ملحوظة : علامة الـbacktick مكانها تحت حرف الـ Esc و بالتحديد هي نفس زرار حرف الـ **ذ**

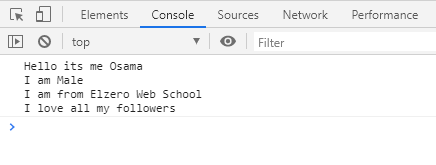
- لو احتاجنا نضيف متغير داخل الـmultiline string .. قبل الـES6 كنا بنحتاج نعمل concatenation بالشكل ده

const GENDER = "Male";

let oldWay = "Hello its me Osama\nI am "+ GENDER +"\nI am from Elzero Web School\nI love all my followers";

console.log(oldWay)

النتيجة



بعد الـES6 الموضوع بقى ابسط .. فبدلا من الـconcatenation هنضيف المتغير في الـmultiline string داخل placeholder بالشكل ده ${ variable } .. مثال

let newWay = `Hello its me Osama

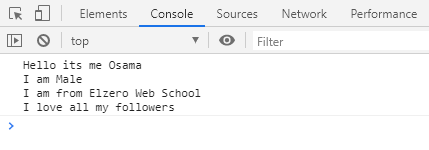
I am ${GENDER}

I am from Elzero Web School

I love all my followers`;

console.log(newWay)

النتيجة



- لو عايزين مثلا نعمل if condition فالـmultiline string على سبيل المثال نضيف كلمة MR لو الـGender كان Male بينما لو كان Female يبقى نضيف كلمة MS

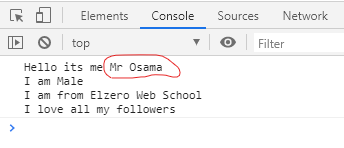
باستخدام الـif condition المختصرة قبل الـES6 كنا بنعمل concatenation بالشكل ده

const GENDER = "Male";

let oldWay = "Hello its me "+ (GENDER === "Male"? "Mr" : "Ms") +" Osama\nI am "+ GENDER +"\nI am from Elzero Web School\nI love all my followers";

console.log(oldWay)

النتيجة



بينما بعد الـES6 هنستغنى عن الـconcatenation كالعادة و هنحط الـcondition داخل الـplaceholder مباشرة بالشكل ده

let newWay = `Hello its me ${GENDER === "Male"? "Mr" : "Ms"} Osama

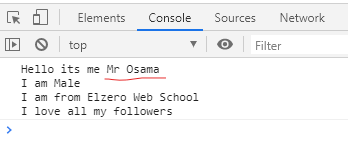
I am ${GENDER}

I am from Elzero Web School

I love all my followers`;

console.log(newWay)

النتيجة



Spread Operator

- الـspread operator من الاضافات المميزة فالـES6 و وظيفته انه بقوم بتفكيك عناصر الـarray

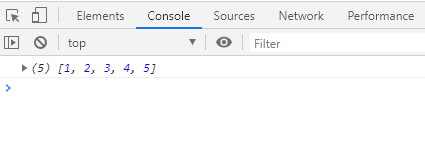
- قبل الـES6 .. لو كان عندنا 2 arrays و عايزين ندمجهم كنا بنعمل كدا

let arrayOne = [1, 2, 3];

let arrayTwo = [4, 5];

console.log(arrayOne.concat(arrayTwo))

النتيجة



بعد الـES6 .. نقدر ندمج الـarrays بشكل اسهل و ابسط بالشكل ده

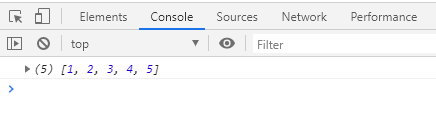
let arrayOne = [1, 2, 3];

let arrayTwo = [4, 5];

let allArrays = [...arrayOne, ...arrayTwo]

console.log(allArrays)

النتيجة



و تفكيك الـarray بالشكل ده ليه مميزات كتير منها اننا نقدر نستخدم عناصر الـ array كـarguments لما ننادي على function .. مثال

function sum(x, y, z){

    return x + y + z;

}

let myNumbers = [1, 2, 3]

console.log( sum(...myNumbers) )

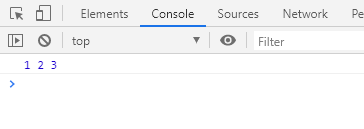
النتيجة 6

و عشان نفهم اللي حصل نشوف الـvalues الخاصة بالـmyNumbers array في الـconsole بعد ما فككنا الـarray

let myNumbers = [1, 2, 3]

console.log(...myNumbers)

النتيجة



- مثال تاني على دمج قيم array داخل array أخر

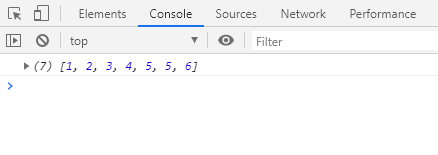
let arrayOne = [1, 2, 3];

let arrayTwo = [4, 5];

let custom = [1, 2, 3, ...arrayTwo, 5, 6];

console.log(custom)

النتيجة



- الـspread operator بيحل مشكلة الـcopy & paste لما نخلي 2 arrays متشابهين و هنفهم ده بمثال

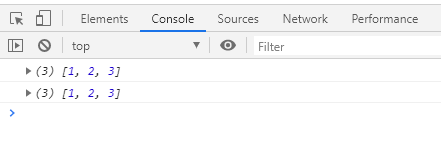
let arrayOne = [1, 2, 3];

let arrayTwo = arrayOne;

console.log(arrayOne)

console.log(arrayTwo)

النتيجة فالـconsole



طيب فين المشكلة ؟

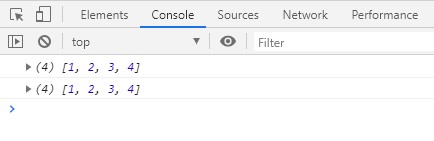
المشكلة هي اننا لو ضيفنا value جديدة للـarrayTwo هنلاقي الـvalue اتضافت للـarrayOne كمان بالشكل ده

arrayTwo.push(4)

console.log(arrayOne)

console.log(arrayTwo)

النتيجة فالـconsole



و الحل هنا اننا نستخدم الـspread operator بالشكل ده

let arrayOne = [1, 2, 3];

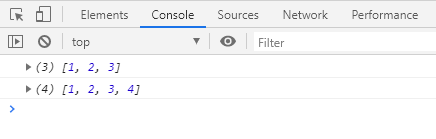
let arrayTwo = [...arrayOne];

arrayTwo.push(4)

console.log(arrayOne)

console.log(arrayTwo)

النتيجة



- لو عندنا array of numbers و عايزين نجيب اصغر رقم باستخدام الـMath.min

let arrayOne = [1, 2, 3, 100, -10, 500];

console.log( Math.min(arrayOne) )

النتيجة هتكون NaN لأن الـMath.min محتاجة مجموعة ارقام وليس array عشان تقارن بينهم و تـreturn اصغر رقم و بالتالي لازم نستخدم الـspread operator عشان نفكك الـarray بالشكل ده

let arrayOne = [1, 2, 3, 100, -20, -10, 500];

console.log( Math.min(...arrayOne) )

النتيجة -20

Default Parameter

- الـDefault Parameter موجود في كل لغات البرمجة و فكرته قايمة على اننا لما نعمل function ليها parameters نضيف default values للـparameters بحيث تتمرر للـfunction بشكل تلقائي لو نسينا نمرر values لما ننادي على الـfunction

- قبل الـES6 .. لو عندنا function ليها parameters بالشكل ده

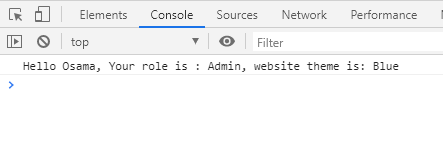
function showMyInfo(username, role, theme){

    return `Hello ${username}, Your role is : ${role}, website theme is: ${theme}` ;

}

console.log( showMyInfo('Osama', 'Admin', 'Blue') )

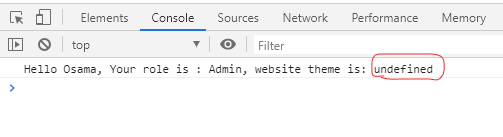
النتيجة في الـconsole



لو نسينا نمرر value لما ننادي الـfunction بالشكل ده

console.log( showMyInfo('Osama', 'Admin') )

النتيجة هتكون



فكنا بنعمل default values للـparameters بالـif conditions المختصرة بالشكل ده

function showMyInfo(username, role, theme){

    let theUserName = ( username === undefined ? 'Default username' : username );

    let theRole = ( role === undefined ? 'Default role' : role );

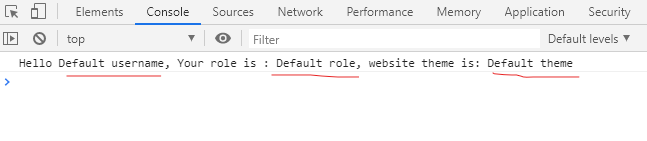
    let theTheme = ( theme === undefined ? 'Default theme' : theme );

    return `Hello ${theUserName}, Your role is : ${theRole}, website theme is: ${theTheme}`;

}

console.log( showMyInfo() );

النتيجة



أو كنا بنستخدم طريقة الـtruthy falsey بالشكل ده

function showMyInfo(username, role, theme){

    let theUserName = username || 'Default username';

    let theRole = role || 'Default role';

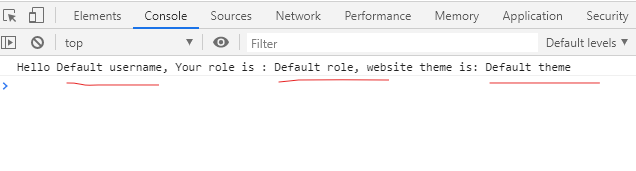
    let theTheme = theme || 'Default theme';

    return `Hello ${theUserName}, Your role is : ${theRole}, website theme is: ${theTheme}`;

}

console.log( showMyInfo() )

النتيجة مش هتتغير



لكن بعد الـES6 الموضوع بقى أبسط .. فبقينا نعمل الـDefault Values للـparameters بالشكل ده

function showMyInfo(username = 'Default username', role = 'Default role', theme = 'Default theme'){

    return `Hello ${username}, Your role is : ${role}, website theme is: ${theme}`;

}

و نجرب ننادي على الـfunction بحالات مختلفة

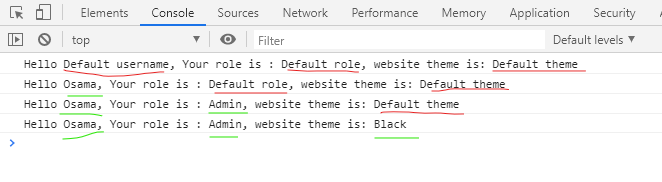
console.log( showMyInfo() )

console.log( showMyInfo('Osama') )

console.log( showMyInfo('Osama', 'Admin') )

console.log( showMyInfo('Osama', 'Admin', 'Black') )

النتيجة



Rest Parameter

- الـrest parameters هي خاصية في الـES6 بتربط مجموعة values داخل array عكس الـspread operator اللي كان بيفكك عناصر الـarray

لو عندنا function بسيطة بالشكل ده

function sum(x, y){

    return x + y;

}

console.log( sum(2, 5) )

النتيجة هتكون 7

و لو ضيفنا argument ثالث بالشكل ده

console.log( sum(2, 5, 6) )

النتيجة مش هتتغير و هتطلع 7 .. لأنك مش محدد فالـfunction غير 2 parameters و بالتالي المتصفح مش هيفهم الـargument الثالث و هنا بتيجي فايدة خاصية الـrest parameters اللي بتخلي عدد الـparameters مرن حسب حاجة المستخدم بحيث ان المستخدم لا يتقيد بعدد معين من الـparameters

مثال

function showMyInfo(a, b, c, ...MyParams){

    console.log('Param a', a);

    console.log('Param b', b);

    console.log('Param c', c);

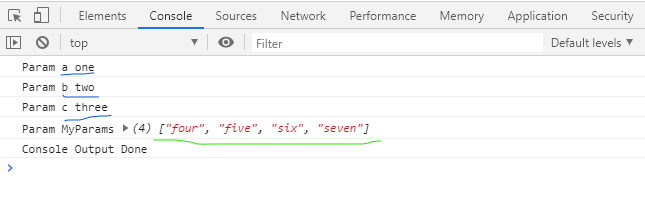
    console.log('Param MyParams', MyParams);

    console.log('Console Output Done')

}

showMyInfo('one', 'two', 'three', 'four', 'five', 'six', 'seven')

النتيجة



اللي حصل هنا ان أي parameter المستخدم يضيفه بعد الـparameters الثالث .. يتم اضافته داخل array

ملحوظة : الـrest parameters operator لازم مكانه يكون اخر parameter .. يعني غير مقبول اننا نعمل كدا

function showMyInfo( ...MyParams, a, b, c)

- هنعمل مثال تاني اكثر توضيحا و فكرته عبارة عن function بتقبل اي عدد من الـparameters و لما ننادي الـfunction يتم جمع قيم الـparameters

function addAll(...MyParams){

    let MyNumber = 0;

    for(param of MyParams){

        MyNumber += param

    }

    return MyNumber

}

console.log( addAll(1, 2, 3) )

ألنتيجة هتكون 6

و لو عملت كدا

console.log( addAll(1, 2, 3, 10, 50, 100) )

النتيجة هتكون 166

New **String** Methods - startsWith, endsWith

- في الـES6 تم اضافة methods جديدة للتعامل مع الـstring منها :

**startsWith**

دي method بتشيك اذا كان الـstring بيبدأ بـtext معين او لأ و بترجع اما true او false و الـsyntax الخاص بيها شكله كدا

// yourString.startsWith('search string', start position)

و الـposition هنا اختياري و by default بيكون 0 الا اذا حددت position معين المتصفح يبدأ منه

مثال

let x = 'Elzero Web School';

console.log( x.startsWith('El') )

النتيجة true

و لو عملنا كدا

let x = 'Elzero Web School';

console.log( x.startsWith('Web') )

النتيجة false

بينما لو عملت كدا

let x = 'Elzero Web School';

console.log( x.startsWith('Web', 7) )

النتيجة true .. لأننا حددنا ان الـsearch يبدا من الـposition رقم 7

**endsWith**

دي method بتشيك اذا كان الـstring بينتهي بـtext معين او لا و بترجع اما true اوfalse و الـsyntax الخاص بيها كدا

// yourString.endsWith('search string', end position)

و الـposition هنا اختياري و by default بيكون الـ**length** بتاع الـstring

مثال

let x = 'Video - Learn How To Learn';

console.log(x.endsWith('Learn'))

النتيجة true

let x = 'Video - Learn How To Learn';

console.log( x.endsWith('To') )

النتيجة false ..بينما لو عملت كدا

let x = 'Video - Learn How To Learn';

console.log( x.endsWith('To', 20) )

النتيجة true

New **String** Methods - Includes, Repeat

- لسه مكملين في الـstring methods و عندنا

**Includes**

و دي method بتشيك اذا كان الـstring بيحتوي على text معين ام لا و بترجع true او false و الـsyntax الخاص بيها كدا

// yourString.endsWith('search string', **start** position)

و الـposition هنا اختياري و by default بيكون 0 الا اذا حددت position معين المتصفح يبدأ منه

مثال

let x = 'Elzero Web School';

console.log( x.includes('Web') )

النتيجة true بينما.. لو عملت كدا

let x = 'Elzero Web School';

console.log( x.includes('Web', 10) )

النتيجة false ..بينما لو عملت كدا

let x = 'Elzero Web School';

console.log( x.includes('eb', 8) )

النتيجة true

**Repeat**

و دي method بتكرر الـstring لعدد معين من المرات انت بتحدده و الـsyntax الخاص بيها شكله كدا

// yourString.repeat(count)

مثال

let x = 'Elzero Web School';

console.log(x.repeat(4))

النتيجة

Elzero Web SchoolElzero Web SchoolElzero Web SchoolElzero Web School

De structuring Object Part 1

- الـDe structuring خاصية في الـES6 بتمكنك انك تعمل extract للبيانات اللي في array او object و تخزنها في متغير و بالتالي تستخدمها بشكل اسهل و ابسط .. و هنبدأ نتكلم عن الـDe structuring في الـobject

لو عندنا object بالشكل ده

var user = {

    name: 'Osama',

    gender: 'Male',

    age: 36,

    city: 'Cairo',

    website: 'Elzero'

}

لو عايزين نـaccess الـname و الـgender و الـage كنا بنعمل كدا

var u = user.name;

var g = user.gender;

var a = user.age;

console.log(`My Name is: ${u}, I am ${g}, My Age is ${a}`)

النتيجة

My Name is: Osama, I am Male, My Age is 36

اللي عملناه ده اسمه De structuring و لكن الطريقة دي طويلة .. في الـES6 الـDe structuring بقى ابسط بالشكل ده

let {name, gender, age} = user

console.log(`My Name is: ${name}, Iam ${gender}, My Age is ${age}`)

النتيجة

My Name is: Osama, I am Male, My Age is 36

ملحوظة : في الـobject de structuring لازم تسمي الـvariables بنفس أسامي الـkeys في الـobject .. مثال

var user = {

    name: 'Osama',

    gender: 'Male',

    age: 36,

    city: 'Cairo',

    website: 'Elzero'

}

let {name, gender, user\_age} = user

console.log(`My Name is: ${name}, I am ${gender}, My Age is ${user\_age}`)

النتيجة

My Name is: Osama, I am Male, My Age is **undefined**

و لو كنا عاملين متغيرات باسم name و gender و age بالشكل ده

var user = {

    name: 'Osama',

    gender: 'Male',

    age: 36,

    city: 'Cairo',

    website: 'Elzero'

}

let name = 'Default';

let gender = 'Default';

let age = 'Default';

و عملنا De structuring بالشكل ده

const {name, gender, age} = user;

console.log(`My Name is: ${name}, I am ${gender}, My Age is ${age}`)

.

النتيجة هتكون Identifier has already been declared لأن الـlet و الـconst مش بيقبلو الـredeclaration

فالحل اننا نعمل assignment باننا نعمل الـDe structuring داخل اقواس ( ) بالشكل

( {name, gender, age} = user );

console.log(`My Name is: ${name}, I am ${gender}, My Age is ${age}`)

النتيجة

My Name is: Osama, I am Male, My Age is 36

- و احنا بنعمل De structuring للـobject .. لو ضيفنا متغير مش موجود في الـobject بالشكل ده

( {name, gender, age, theme} = user );

console.log(`My Name is: ${name}, I am ${gender}, My Age is ${age}, My Theme Color is: ${theme}`)

النتيجة هتكون

My Name is: Osama, I am Male, My Age is 36, My Theme Color is: **undefined**

و ده بيتم معالجته باننا نعمل Default value للمتغير اللي مش موجود في الـobject بالشكل ده

( {name, gender, age, theme = 'Default'} = user );

console.log(`My Name is: ${name}, I am ${gender}, My Age is ${age}, My Theme Color is: ${theme}`)

النتيجة هتكون

My Name is: Osama, I am Male, My Age is 36, My Theme Color is: Default

و لو ضيفنا بيانات الـtheme للـobject بالشكل ده

var user = {

    name: 'Osama',

    gender: 'Male',

    age: 36,

    city: 'Cairo',

    website: 'Elzero',

    theme: 'red'

}

هيحصل override على الـDefault value و النتيجة هتكون

My Name is: Osama, I am Male, My Age is 36, My Theme Color is: red

De structuring Object Part 2

- لسه مكملين في الـobject De structuring .. لما عملنا De structuring للـobject كنا مجبرين اننا نسمي المتغيرات بنفس اسامي الـkeys اللي في الـobject بالشكل ده

var user = {

    name: 'Osama',

    gender: 'Male',

    age: 36,

    city: 'Cairo',

    website: 'Elzero',

    theme: 'red'

}

const {name, gender, age, theme = 'Default'} = user;

لو حبيت اغير في اسم متغير الـname في الـDe structuring و عملت كدا

const {n, gender, age, theme = 'Default'} = user;

console.log(`My Name is: ${n}, I am ${gender}, My Age is ${age}, My Theme Color is: ${theme}`)

النتيجة

My Name is: **undefined**, I am Male, My Age is 36, My Theme Color is: red

و الحل هنا اننا لو عايزين نسمي المتغيرات في الـobject De structuring باسامي مختلف عن اسامي الـobject keys اننا نسميها بالـsyntax ده

const {name: theName, gender: theGender, age: theAge, theme: theTheme = 'Default'} = user;

console.log(`My Name is: ${theName}, I am ${theGender}, My Age is ${theAge}, My Theme Color is: ${theTheme}`)

النتيجة

My Name is: Osama, I am Male, My Age is 36, My Theme Color is: red

- مثال تاني متقدم اكتر .. لو افترضنا اننا عندنا object اسمه langs داخل الـuser object بالشكل ده

var user = {

    name: 'Osama',

    gender: 'Male',

    age: 36,

    city: 'Cairo',

    website: 'Elzero',

    theme: 'red',

    langs: {

        html: '70%',

        css: '80%',

        js: '10%',

    }

}

و عملنا كدا

const { name, gender, langs } = user;

console.log(`My Name is: ${name}, I am ${gender}, My Languages ${langs}`)

النتيجة هتكون

My Name is: Osama, I am Male, My Languages [object Object]

و الحل هنا اننا نـaccess الداتا في الـlangs بالطريقة دي

const { name, gender, langs } = user;

console.log(`My Name is: ${name}, I am ${gender}, My HTML Progress is: ${langs.html}`)

و النتيجة هتكون

My Name is: Osama, I am Male, My HTML Progress is: 70%

او اننا نعمل كدا

const { name, gender, langs: {html, css, js} } = user;

console.log(`My Name is: ${name}, I am ${gender}, My HTML Progress is: ${html}`)

و النتيجة مش هتتغير

My Name is: Osama, I am Male, My HTML Progress is: 70%

و لو ضيفنا متغير لـkey مش موجود في الـobject بالشكل ده

const { name, gender, langs: {html, css, js, python} } = user;

console.log(`My Name is: ${name}, I am ${gender}, My HTML Progress is: ${html} And My Python Progress is ${python}`)

النتيجة هتكون

My Name is: Osama, I am Male, My HTML Progress is: 70% And My Python Progress is undefined

و الحل اننا نعمل default value زي ما عملنا قبل كدا بالشكل ده

const { name, gender, langs: {html, css, js, python = '10%'} } = user;

console.log(`My Name is: ${name}, I am ${gender}, My HTML Progress is: ${html} And My Python Progress is ${python}`)

و النتيجة هتكون

My Name is: Osama, I am Male, My HTML Progress is: 70% And My Python Progress is 10%

و لو عملنا key للـpython في الـlangs object .. هيحصل override على الـdefault value

- لو عايزين نعمل De structuring للـlangs object فقط بدلا من الـuser object كله .. يبقى هنعمل كدا

const { html, css, js } = user.langs

console.log(`${html}, ${css}, ${js}`)

النتيجة

70%, 80%, 10%

De structuring Array Part 1

- الـarray De structuring فكرته زي الـobject De structuring و اسهل و ابسط و نفهمه بمثال

const food = ["Burger", "Pizza", "Chicken"]

const [one, two, three] = food

كل متغير من one و two و three بياخد الـvalue اللي بتوازيه في الـfood array .. فلو عملنا كدا

console.log(`The Food I Love Is ${one}, ${two}, ${three}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken

و لو المتغيرات one و two و three موجودين بالفعل قبل ما نعمل De structuring بالشكل ده

const food = ["Burger", "Pizza", "Chicken"]

let one = "Default";

let two = "Default";

let three = "Default";

const [one, two, three] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}`)

النتيجة هتكون

Identifier 'one' has already been declared

لأن مينفعش نعمل redeclaration مع الـlet و انما ممكن re assignment بالشكل ده

[one, two, three] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken

أو اننا نعمل الـde structuring داخل ( ) بالشكل ده

([one, two, three] = food)

console.log(`The Food I Love Is ${one}, ${two}, ${three}`)

و النتيجة كما هي

و لو عملنا متغير رابع في الـde structuring بالشكل ده

const food = ["Burger", "Pizza", "Chicken"]

const [one, two, three, four] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}, ${four}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken, undefined

و الحل اننا نعمل default value

const [one, two, three, four = "Default"] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}, ${four}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken, Default

و لو ضيفنا عنصر رابع الـfood array

const food = ["Burger", "Pizza", "Chicken", "Meat"]

const [one, two, three, four = "Default"] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}, ${four}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken, Meat

- في الـarray de structuring نقدر نعمل skip لـvalues معينة باننا نسيب مكان المتغير فاضي بالشكل ده

const food = ["Burger", "Pizza", "Chicken", "Meat", "Rice"]

const [one, , three, four] = food

console.log(`The Food I Love Is ${one}, ${three}, ${four}`)

النتيجة

The Food I Love Is Burger, Chicken, Meat

De structuring Array Part 2

- لسه مكملين فالـarray de structuring و هناخد شوية امثلة advanced اكتر

هنشتغل على مثال الـfood array بس هنعمل nested array داخل الـfood array بالشكل ده

const food = [ "Burger", "Pizza", "Chicken", ["Apple", "Banana", "Mango"] ]

و الـde structuring هنا مش هيختلف غير اننا هنعمل array de structuring داخل الـfood array de structuring كدا

const [ one, two, three, [ fr1, fr2, fr3 ] ] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}. The Fruits I Love is ${fr1}, ${fr2}, ${fr3}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken. The Fruits I Love is Apple, Banana, Mango

- ممكن نستخدم الـRest operator في الـarray de structuring بالشكل ده

const food = [ "Burger", "Pizza", "Chicken", "Meat", "Rice", "Pasta" ]

const [ one, two, three, ...otherFood ] = food

console.log(`The Food I Love Is ${one}, ${two}, ${three}. Other Food I Love is ${otherFood}`)

النتيجة

The Food I Love Is Burger, Pizza, Chicken. Other Food I Love is Meat,Rice,Pasta

De structuring Array - Swap Variables

- الـswap variables هو تبديل الـvalues ما بين الـvariable .. مثال

لو عندنا 2 variables بالشكل ده

var book = "Video";

var video = "Book";

console.log(`book is ${book} and video is ${video}`)

النتيجة

book is Video and video is Book

لو عايزين نبدل الـvalues ما بين الـbook variable و الـvideo variable هنستخدم خاصية الـswap variable بالشكل ده

var book = "Video";

var video = "Book";

[book, video] = [video, book]

console.log(`book is ${book} and video is ${video}`)

النتيجة

book is Book and video is Video

De structuring Mixed Content

- الدرس ده هنعمل مثال متقدم على الـde structuring و هيكون عبارة عن array و object داخل object بالشكل ده

const user = {

    username: 'Osama',

    city: 'Cairo',

    skills: {

        html: '80%',

        css: '85%',

        js: ['Vuejs', 'Reactjs', 'Angularjs']

    }

};

هنعمل de structuring للـuser object و الـobjects و الـarrays اللي جواه بالشكل ده

const { username, city, skills: { html, css, js: [Vuejs, Reactjs, Angularjs] } } = user;

console.log(`My name is ${username}, and I live in ${city}`);

console.log(`My HTML skills progress is ${html}, and CSS is ${css}`);

console.log(`I have knowledge in JS frameworks like ${Vuejs}, ${Reactjs}, ${Angularjs}`)

النتيجة

My name is Osama, and I live in Cairo

My HTML skills progress is 80%, and CSS is 85%

I have knowledge in JS frameworks like Vuejs, Reactjs, Angularjs

Destructuring Function Parameters

- اخر درس عندنا في الـde structuring هو الـde structuring الخاص بالـfunction parameters

بالتطبيق على نفس الـuser object اللي خدناه في الدرس اللي فات

const user = {

    username: 'Osama',

    city: 'Cairo',

    skills: {

        html: '80%',

        css: '85%',

        js: ['Vuejs', 'Reactjs', 'Angularjs']

    }

};

لو عندنا function اسمها showMyInfo هتعرض الداتا اللي في الـuser object و عايزين نمرر الـuser object كـparameter بس نعمله de structuring .. بكل بساطة الـde structuring هيكون بالشكل ده

function showMyInfo( { username, city, skills: { html, css, js: [Vuejs, Reactjs, Angularjs] } } ) {

    console.log(`My name is ${username}, and I live in ${city}`);

    console.log(`My HTML skills progress is ${html}, and CSS is ${css}`);

    console.log(`I have knowledge in JS frameworks like ${Vuejs}, ${Reactjs}, ${Angularjs}`)

}

و لما ننادي على الـfunction .. هنمرر الـ user object كـargument بحيث ان البيانات اللي فيه يحصلها de structuring

بالشكل ده

showMyInfo(user)

النتيجة

My name is Osama, and I live in Cairo

My HTML skills progress is 80%, and CSS is 85%

I have knowledge in JS frameworks like Vuejs, Reactjs, Angularjs

Enhanced Object Literals

- الدرس ده هنتناول بعض التحسينات على استخدام الـobject بعد اصدار الـES6

1- لو عندنا مثال بالشكل ده

const username = "Osama";

const website = "Elzero.org";

const user = {

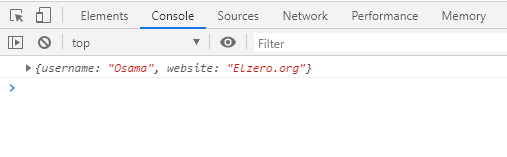
    username: username,

    website: website

};

console.log(user)

النتيجة هتكون



التحديث في الـES6 هو ان لو اسم الـproperty في الـobject متشابه مع اسم المتغير يبقى ممكن تلغي الـvalue الخاصة بالـproperty دي في الـobject بالشكل ده

const username = "Osama";

const website = "Elzero.org";

const user = {

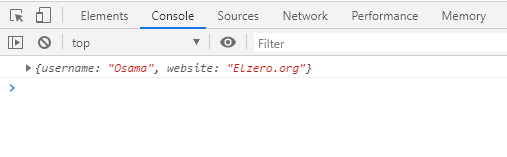
    username,

    website

};

console.log(user)

النتيجة مش هتتغير



2- لو عندنا object فيه property عبارة عن function بالشكل ده

const user = {

    myMethod: function(param1){

        return param1

    }

};

console.log( user.myMethod('Hello From Parameter') )

النتيجة

Hello From Parameter

التحديث في الـES6 هو اننا نقدر نلغي كلمة function بالشكل ده

const user = {

    myMethod(param1){

        return param1

    }

};

console.log( user.myMethod('Hello From Parameter') )

النتيجة مش هتتغير

Hello From Parameter

**معلومة** : اصدار الـES6 قدم تسهيلات لتقليل استخدام كلمة function في الجافاسكريبت

3- لو عندنا object بسيط بالشكل ده

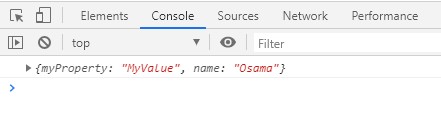
const myObject = { myProperty: 'MyValue' };

و حبينا نضيف property للـobject من خارج الـobject كنا بنعمل كدا

myObject['name'] = 'Osama'

console.log( myObject )

النتيجة



و عشان نخلي الـproperty اللي هنضيفها تكون dynamic بمعنى انها مرتبطة بمتغير خارجي .. كنا بنعمل كدا

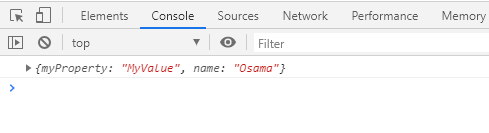
const myVariable = 'name';

const myObject = { myProperty: 'MyValue' };

myObject[myVariable] = 'Osama'

console.log( myObject )

النتيجة



بعد الـES6 .. اضافة dynamic property للـobject بقى اسهل و ابسط .. بالشكل ده

const myVariable = 'name';

const myObject = {

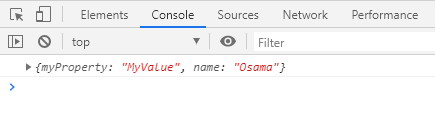
    myProperty: 'MyValue',

    [myVariable]: 'Osama'

};

console.log( myObject )

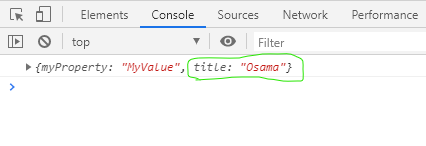
النتيجة



لو غيرنا الـvalue الخاصة بالمتغير myVariable بالشكل ده

const myVariable = 'title';

النتيجة هتكون



Set vs Array

- الـSet من ضمن الاضافات في الـES6 و الـSet شبه الـarray مع وجود اختلافات بينهم منها ان الـSet مش بيقبل values متكررة بمعنى ان كل الـvalues اللي فيه unique .. و الـSet شبه الـ object و لكنه بيختلف عن الـobject العادي بانه ليس له keys او properties و الـsyntax الخاص بالـSet شكله كدا

// var yourVariable = new Set( iterable )

و المقصود بكلمة iterable هو اي value ممكن نعمل عليها loop زي الـarray و الـstring

مثال

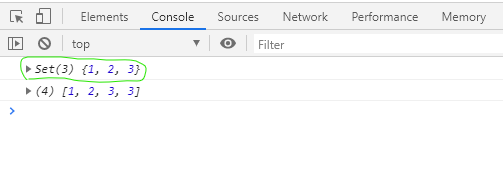
let myArray = [1, 2, 3, 3];

let myBooks = new Set( myArray );

console.log( myBooks );

console.log( myArray );

النتيجة



نلاحظ ان رقم 3 متكررش في الـSet و ده لان الـSet مش بتقبل داتا متكررة ( duplicated values )

- في الـarray نقدر نـaccess اي value فيه من خلال الـindex بينما ده غير متاح في الـSet .. مثال

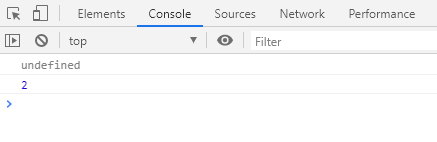
let myArray = [1, 2, 3];

let myBooks = new Set( myArray );

console.log( myBooks[1] );

console.log( myArray[1] );

النتيجة



- نقدر نحول الـarray لـSet زي ما عملنا

let myArray = [1, 2, 3];

let myBooks = new Set( myArray );

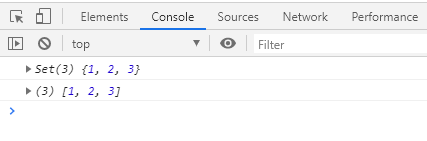
بينما لو عايزين نحول الـSet لـarray هنستخدم الـspread operator .. مثال

let myBooks = new Set( [1, 2, 3] );

console.log( myBooks );

console.log( [...myBooks] );

النتيجة



Set Methods

- هنبدأ نتكلم عن الـmethods الخاصة بالـSet

1- عشان نضيف value في الـSet بنستخدم method اسمه add .. مثال

let myBooks = new Set().add('one').add('two').add('three');

console.log( myBooks );

او كدا

let myBooks = new Set();

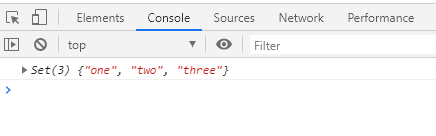
myBooks.add('one');

myBooks.add('two');

myBooks.add('three');

console.log( myBooks );

النتيجة في الحالتين واحدة

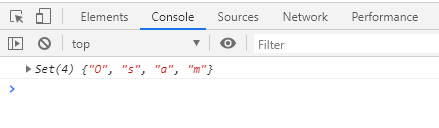


- لو جربنا نضيف value للـSet بالشكل ده

let myBooks = new Set('Osama');

console.log( myBooks );

النتيجة



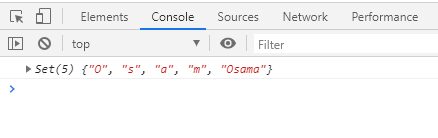
نلاحظ ان حروف Osama اتفككت لان الـSet بتقبل iterable value ما بين الاقواس فبتعمل iteration للـvalue اللي بين الاقواس و حرف الـa اتحذف لأنه اتكرر .. بينما بنستخدم الـadd method لو عايزين نضيف value من غير ما الـSet تعمل iteration عليها .. مثال للتوضيح

let myBooks = new Set('Osama');

myBooks.add('Osama');

console.log( myBooks );

النتيجة



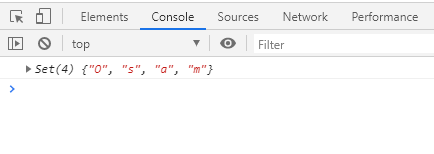
بينما لو عملنا كدا

let myBooks = new Set('Osama');

myBooks.add('O');

console.log( myBooks );

النتيجة هتكون



حرف الـO ما اتضافش لانه متكرر

2- لو عايزين نحذف value في الـSet بنستخدم method اسمه delete .. مثال

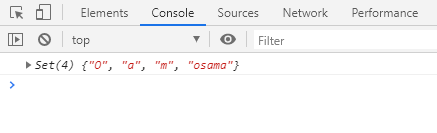
let myBooks = new Set('Osama');

myBooks.add('osama');

myBooks.delete('s');

console.log( myBooks );

النتيجة



3- لو عايزين نشيك على value في الـSet موجودة ولا لأ .. بنستخدم method اسمه has و النتيجة هتكون اما true او false

مثال

let myBooks = new Set('Osama');

myBooks.add('osama');

console.log( myBooks.has('osama') );

النتيجة هتكون true .. بينما لو عملنا كدا

console.log( myBooks.has('Osama') );

النتيجة هتكون false لأننا بنبحث Osama و اول حرف capital .. بينما osama في الـSet اول حرف منها small

4- لو عايزين نعرف عدد الـvalues في الـSet بنستخدم method اسمها size .. مثال

let myBooks = new Set('Osama');

myBooks.add('osama');

console.log( myBooks.size );

النتيجة هتكون 5

5- لو عايزين نحذف كل الـvalues في الـSet بنستخدم method اسمه clear .. مثال

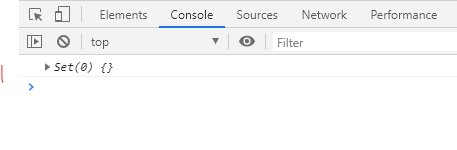
let myBooks = new Set('Osama');

myBooks.add('osama');

myBooks.clear()

console.log( myBooks );

النتيجة



Map vs Object

- الـMap هو data type زي الـobject بيتكون من keys و values لكن الفرق بين الـMap و الـobject هو ان الـMap بيقبل كل انواع الـdata types في الـkeys على عكس الـobject

- قبل الـES6 كنا بنعمل الـMap باننا نعمل object فاضي و نملاه بالـvalues .. مثال

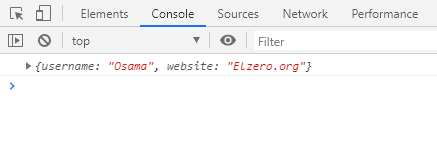
var user = {}

user['username'] = "Osama";

user.website = "Elzero.org";

console.log(user)

النتيجة



بعد الـES6 .. عمل الـMap بقى اسهل بالشكل ده

let myMap = new Map();

myMap.set("10", "string").set(10, "number").set(false, "boolean").set({a: 1, b: 2}, "object").set(function doSomething() {}, "function")

console.log(myMap)

او كدا

let myMap = new Map();

myMap.set("10", "string")

myMap.set(10, "number")

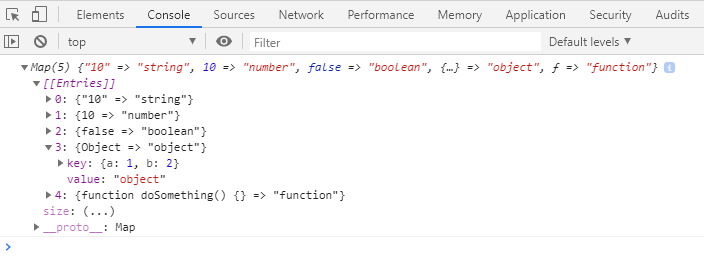
myMap.set(false, "boolean")

myMap.set({a: 1, b: 2}, "object")

myMap.set(function doSomething() {}, "function")

console.log(myMap)

النتيجة واحدة في الحالتين



- من ضمن الاختلافات بين الـobject و الـMap .. ان الـobject بيحول الـdata type لكل الـkeys لـstring .. مثال

const myVariable = 'title';

var user = {}

user['username'] = "Osama";

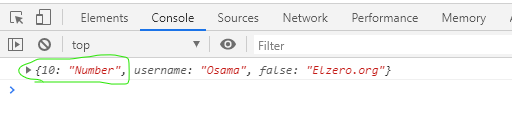
user.false = "Elzero.org";

user['10'] = "String";

user[10] = "Number";

console.log(user)

النتيجة



الـuser[10] تحولت لـstring و بالتالي حصل override و اصبح الـvalue الخاصة بالـ10 هي Number بدلا من String

بينما الـMap بيحتفظ بالـdata types الخاصة بالـkeys مع العلم ان الـMap مش بيقبل values متكررة بمعنىvalues متشابهة بنسبة 100 % .. مثال

let myMap = new Map();

myMap.set("10", "string")

myMap.set(10, "number")

myMap.set(10, "number 2")

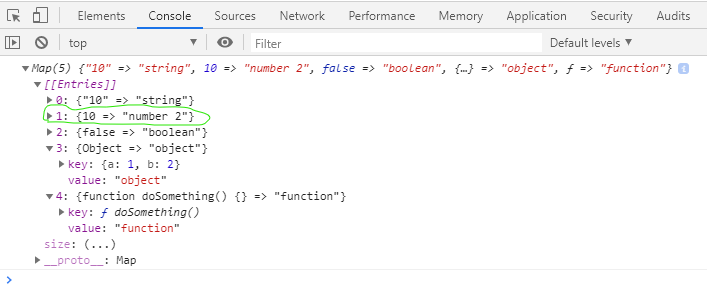
myMap.set(false, "boolean")

myMap.set({a: 1, b: 2}, "object")

myMap.set(function doSomething() {}, "function")

console.log(myMap)

النتيجة



- عندنا syntax ظريف لاضافة الـvalues للـMap بيعتمد على array of arrays بدلا من الـset method بالشكل ده

let myMap = new Map( [

    ["10", "string"],

    [10, "number"],

    [false, "boolean"],

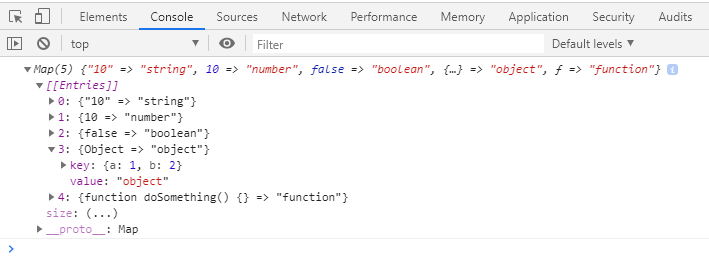
    [{a: 1, b: 2}, "object"],

    [function doSomething() {}, "function"],

] );

console.log(myMap)

النتيجة مش هتتغير



Map Methods

- هنتكلم الدرس ده عن الـMap methods

1- لو عايز اعرف عدد العناصر في الـMap بنستخدم method اسمه size .. مثال

let myMap = new Map( [

    ["10", "string"],

    [10, "number"],

    [false, "boolean"],

    [{a: 1, b: 2}, "object"],

    [function doSomething() {}, "function"],

] );

console.log(`My Map Elements Count Is ${myMap.size}`)

النتيجة 5

2- لو عايز اجيب value خاصة بـkey معين بستخدم method اسمه get .. مثال

let myMap = new Map( [

    ["10", "string"],

    [10, "number"],

    [false, "boolean"],

    [{a: 1, b: 2}, "object"],

    [function doSomething() {}, "function"],

] );

console.log( myMap.get('10') )

النتيجة string

3- لو عايز اشيك على key في الـMap هل موجود ام لا بستخدم method اسمه has و الـmethod ده هيـreturn اما true او false .. مثال

let myMap = new Map( [

    ["10", "string"],

    [10, "number"],

    [false, "boolean"],

    [{a: 1, b: 2}, "object"],

    [function doSomething() {}, "function"],

] );

console.log( myMap.has('false') )

النتيجة false

بينما لو عملت كدا

console.log( myMap.has(false) )

النتيجة هتكون true

4- لو عايز احذف عنصر من الـMap بستخدم method اسمه delete .. مثال

let myMap = new Map( [

    ["10", "string"],

    [10, "number"],

    [false, "boolean"],

    [{a: 1, b: 2}, "object"],

    [function doSomething() {}, "function"],

] );

myMap.delete(false)

console.log( myMap.size )

النتيجة 4

5- لو عايز احذف كل عناصر الـMap بستخدم method اسمه clear .. مثال

let myMap = new Map( [

    ["10", "string"],

    [10, "number"],

    [false, "boolean"],

    [{a: 1, b: 2}, "object"],

    [function doSomething() {}, "function"],

] );

myMap.clear()

console.log( myMap.size )

النتيجة 0

Symbol Part 1

- الـSymbol هو primitive data type في الـES6 و المقصود بكلمة primitive data type انه ليس له methods

و الـSymbol يعتبر unique value بشكل تلقائي و غير قابل للتعديل .. مثال

const mySympol = Symbol()

console.log( typeof(mySympol) )

النتيجة هتكون symbol

و عشان نثبت ان كل symbol يعتبر unique value هنجرب الاتي

const mySympolOne = Symbol("Testing")

const mySympolTwo = Symbol("Testing")

console.log( mySympolOne === mySympolTwo )

النتيجة false

- بنستخدم الـsymbol

1- عشان نمنع اي override في الـobject لو حصل ان property تشابهت مع property تانية

2- عشان نحافظ على الـprivacy لبعض البيانات في الـobject بحيث متظهرش لو حصل loop على الـobject .. مثال

const myNewSympol = Symbol()

const myObject = {

    username: "Osama Elzero",

    website: "Elzero.org",

    [myNewSympol]: "This Is Hidden From Iteration"

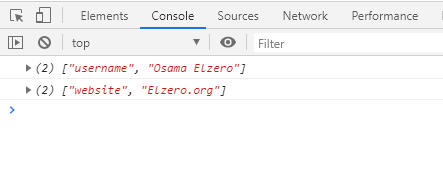
}

for( let val of Object.entries(myObject) ) {

    console.log(val)

}

النتيجة



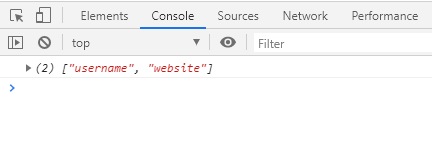
و لو فكرت اطبع الـproperties او الـkeys في الـobject بالشكل ده

console.log( Object.getOwnPropertyNames(myObject) )

او بالشكل ده

console.log( Object.keys(myObject) )

النتيجة ان الـsymbol مش هيظهر



Symbol Part 2

- مكملين في الـsymbol و الدرس ده هنتكلم عن method اسمه for بيمكنا اننا ننشئ symbols

و نـaccess الـglobal symbols اللي عندنا .. مثال

عشان ننشئ symbol بالـfor method بنعمل كدا

const mySympolOne = Symbol.for("Testing"); //create symbol named **Testing** if not exist or return it if exists

console.log(mySympolOne)

النتيجة

Symbol(Testing)

لو جينا و عملنا كدا

const mySympolOne = Symbol.for("Testing"); //create symbol named **Testing** if not exist or return it if exists

const mySympolTwo = Symbol.for("Testing"); //create symbol named **Testing** if not exist or return it if exists

console.log(mySympolOne === mySympolTwo)

النتيجة هتكون true .. لأننا

في المتغير الأول عملنا search على symbol اسمه Testing و بما انه غير موجود .. تم انشاء symbol باسم Testing

بينما في المتغير التاني عملنا search على symbol اسمه Testing و و بما انه موجود .. حصل return للـTesting symbol

و لو عملنا كدا

const mySympolOne = Symbol.for("One"); //create symbol named **One** if not exist or return it if exists

const mySympolTwo = Symbol.for("Two"); //create symbol named **Two** if not exist or return it if exists

console.log(mySympolOne === mySympolTwo)

النتيجة هتكون false

بينما لو عملنا كدا

const mySympolOne = Symbol("Testing");

const mySympolTwo = Symbol("Testing");

console.log(mySympolOne === mySympolTwo)

النتيجة هتكون false .. لأننا بالطريقة دي أنشأنا 2 symbols و الـsymbol بطبيعته يعتبر unique value

- عندنا method تاني خاص بالـsymbols اسمه keyFor و ده بيـreturn الـkey الخاص بالـsymbols

اللي أنشأناها بالـ for method فقط .. مثال

const mySympolTwo = Symbol.for("Two");

console.log(Symbol.keyFor(mySympolTwo))

النتيجة Two

بينما لو عملنا كدا

const mySympol = Symbol("Testing");

console.log( Symbol.keyFor(mySympol) )

النتيجة هتكون undefined

- عندنا method تالت خاص بالـsymbols اسمه description و ده بيـreturn الـdescription الخاص بالـsymbols

زي الـkeyFor لكن **الفرق** انه بيشتغل مع كل الـsymbols

بينما الـkeyFor بيشتغل مع الـsymbols اللي أنشأناها بالـ for method فقط .. مثال

const mySympolTwo = Symbol.for("Two");

const mySympol = Symbol("Testing");

console.log( mySympol.description )

console.log( mySympolTwo.description )

النتيجة

Testing

Two

forEach And Discussions

- الـforEach هي array method بتعمل loop على كل عناصر الـarray و بتنفذ arrow function بتتنفذ على كل عنصر في الـarray

- الـforEach بتاخد parameter وحيد عبارة عن الـarrow function .. و الـarrow function ليها 3 parameters

الأول بيمثل الـvalue

الثاني بيمثل الـindex

الثالث بيمثل الـarray ككل

- الـsyntax بتاع الـforEach method شكله كدا

.forEach( currentValue, index, array )

- لو عندنا array بالشكل ده

const array = ['A', 'B', 'C', 'D',  ,'F'];

و محتاجين نعمل loop عليه .. كنا بنستخدم الطريقة التقليدية و هي الـfor loop بالشكل ده

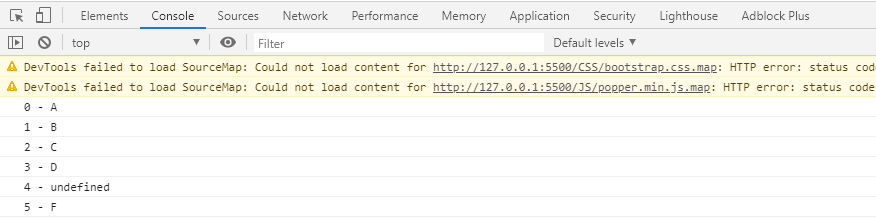
const array = ['A', 'B', 'C', 'D',  ,'F'];

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

    console.log(`${i} - ${array[i]}`)

}

النتيجة



لكن باستخدام الـforEach .. الموضوع بقى أبسط بالشكل ده

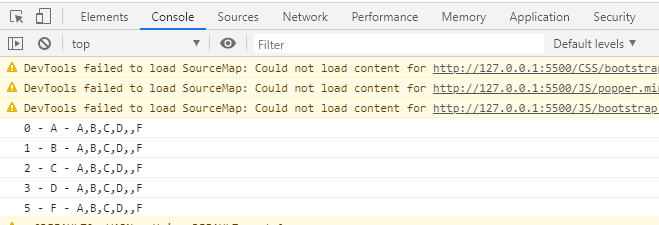
const array = ['A', 'B', 'C', 'D',  ,'F'];

array.forEach( (element, index, all )=> {

    console.log(`${index} - ${element} - ${all}`)

} );

النتيجة



نلاحظ ان الـforEach عملت skip للـvalue الفاضية بدل ما تـreturn undefined و بالتالي عدد الـcallbacks أو عدد مرات تنفيذ الـfunction بيكون أد عدد العناصر الفعلية في الـarray و دي من مميزات الـforEach

مثال

const array = ['A', 'B', 'C', 'D',  ,'F'];

let callbacks = 0;

array.forEach( (element, index, all )=> {

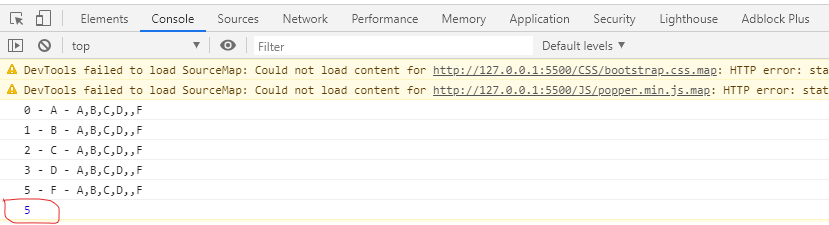
    console.log(`${index} - ${element} - ${all}`);

    callbacks++;

} );

console.log(callbacks)

النتيجة



forEach With Set And Map

- الدرس ده هنتعلم ازاي نستخدم الـforEach مع الـSet و الـMap

**أولا : مع الـSet**

- بما ان مفيش index أو key في الـSet .. اذا الـparameter الأول و الـparameter التاني في الـarrow function هيمثلوا نفس الـvalue

و الـsyntax شكله كدا

.forEach(currentValue, currentValue, Set)

مثال

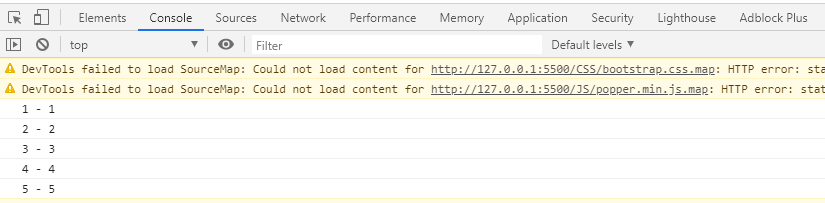
const mySet = new Set( [1, 1, 1, 2, 2, 2, 3, 4, 5] );

mySet.forEach( (item1, item2 )=> {

    console.log(`${item1} - ${item2}`);

} );

النتيجة



**ثانيا : مع الـMap**

- بما ان الـMap فيه keys و values زي الـobject .. اذا الـparameter الأول في الـarrow function هيمثل الـvalue و الـparameter التاني هيمثل الـkey

و الـsyntax شكله كدا

.forEach(value, key, Map)

مثال

const myMap = new Map( [

    [ 'A', 1 ],

    [ 'C', 3 ],

    [ 'B', 2 ],

    [ 'X', {} ]

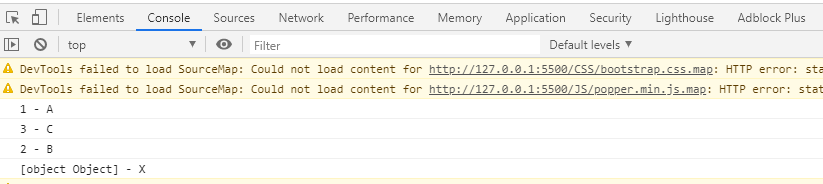
] );

myMap.forEach( (v, k )=> {

    console.log(`${v} - ${k}`);

} );

النتيجة

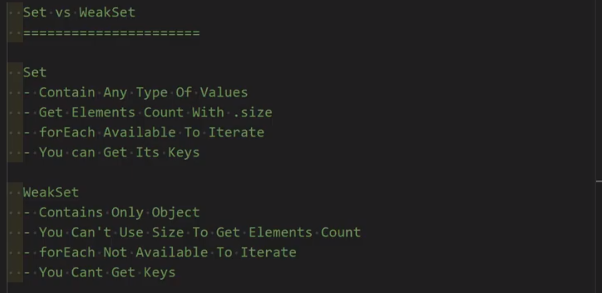


Set vs WeakSet And Garbage Collection

معلومة : الـgarbage collection مش مرتبطة بالجافاسكريبت و انما بالـmemory management في أي لغة برمجة

والـgarbage collection بيمثل البيانات اللي الـlanguage engine بيحذفها من الـmemory بعد انتهاء الغرض منها تماما و بالتالي الضغط على الـmemory يقل

- الـWeakSet شبه الـSet باستثناء بعض الفروقات دي



مثال بسيط على الـSet

let mainObject = {A: 1};

let mySet = new Set([ mainObject ]);

let objectOne = {B: 2};

let objectTwo = {C: 3};

mySet.add(objectOne);

mySet.add(objectTwo);

console.log(mySet.has(objectOne));

console.log(mySet.has(objectTwo));

mySet.add("String");

mySet.add(12);

console.log(mySet.size);

console.log(mySet.keys());

mySet.delete(objectOne);

console.log(mySet.has(objectOne));

objectTwo = null;

console.log(mySet.has(objectTwo));

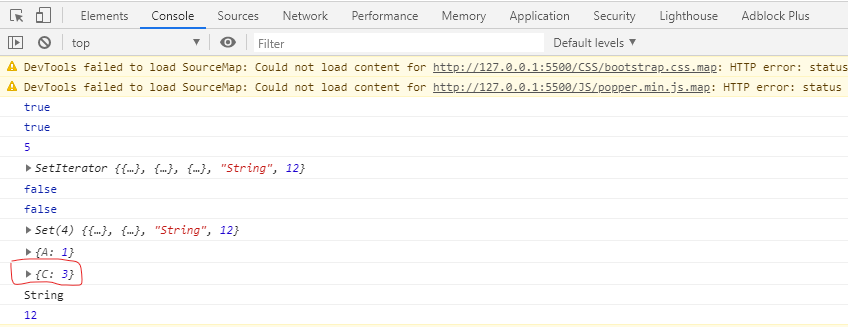
console.log(mySet);

mySet.forEach(v => {

    console.log(v);

});

النتيجة



نلاحظ ان الـobjectTwo مازال موجود بنفس الـvalue غم اننا حذفناه باننا عملناله re-assignment و خلينا الـvalue بتاعته null

نطبق نفس المثال لكن باستخدام الـWeakSet و محاولة اضافة value من أي data type غير الـobject

let myWeakSet = new WeakSet([ mainObject ]);

let objectOne = {B: 2};

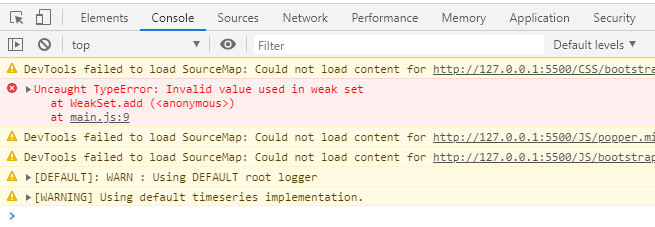
let objectTwo = {C: 3};

myWeakSet.add("String");

myWeakSet.add(12);

console.log(myWeakSet);

النتيجة



مثال أخر على الـWeakSet و استخدام الـsize و الـkeys

let mainObject = {A: 1};

let myWeakSet = new WeakSet([ mainObject ]);

let objectOne = {B: 2};

let objectTwo = {C: 3};

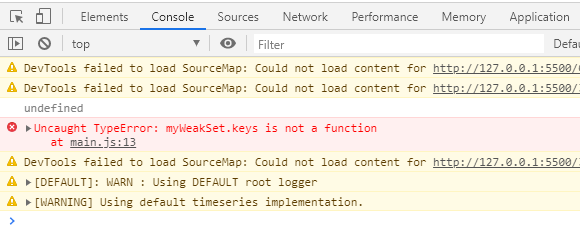
myWeakSet.add(objectOne);

myWeakSet.add(objectTwo);

console.log(myWeakSet.size);

console.log(myWeakSet.keys());

النتيجة



مثال أخر هنجرب فيه حذف الـvalues بالـdelete و الـnull

let mainObject = {A: 1};

let myWeakSet = new WeakSet([ mainObject ]);

let objectOne = {B: 2};

let objectTwo = {C: 3};

myWeakSet.add(objectOne);

myWeakSet.add(objectTwo);

console.log(myWeakSet.has(objectOne));

console.log(myWeakSet.has(objectTwo));

myWeakSet.delete(objectOne);

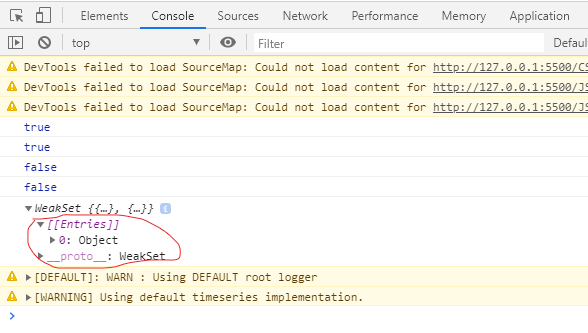
console.log(myWeakSet.has(objectOne));

objectTwo = null;

console.log(myWeakSet.has(objectTwo));

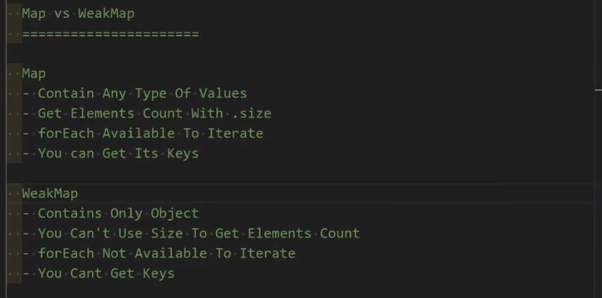
console.log(myWeakSet);

النتيجة .. هنلاقي الـgarbage collection حذف الـobjectTwo على عكس الـSet بالشكل ده

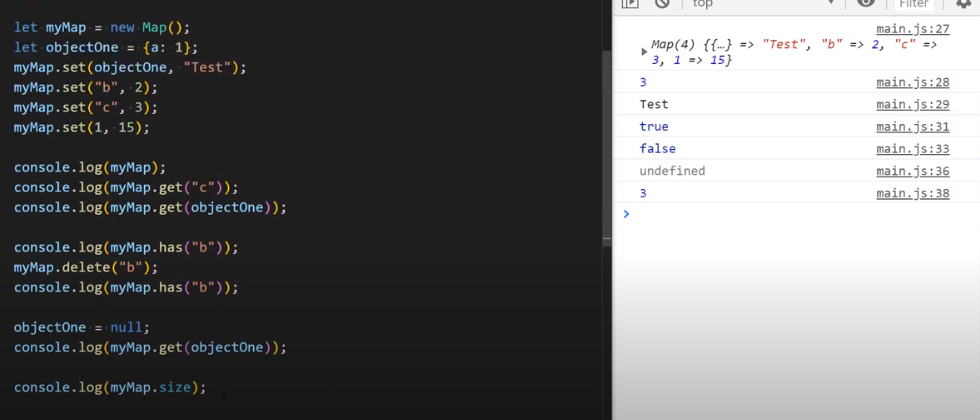


#29 - Map vs WeakMap

- الـWeakMap شبه الـMap باستثناء بعض الفروقات دي



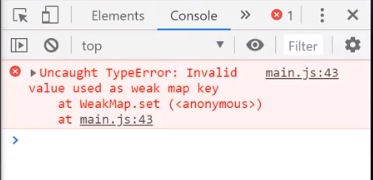
مثال بسيط على الـMap



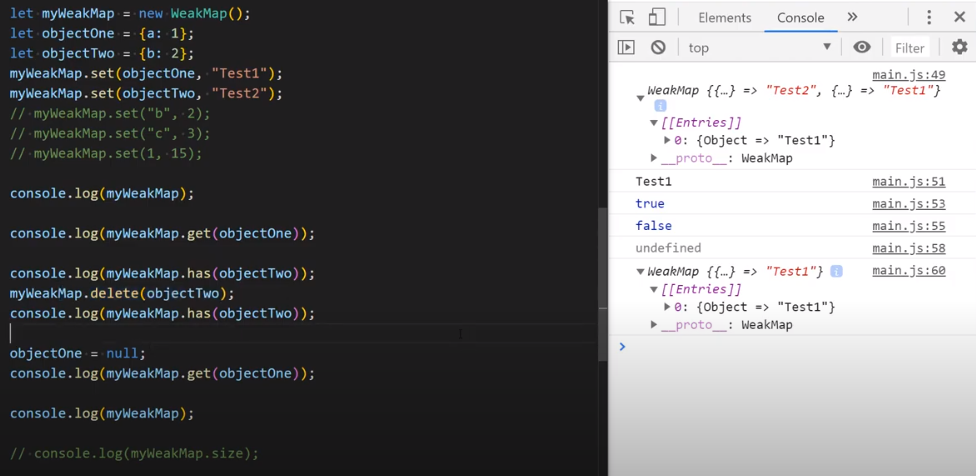
نطبق نفس المثال لكن باستخدام الـWeakMap و محاولة اضافة key من أي data type غير الـobject



النتيجة



مثال أخر على الـWeakMap و استخدام الـset و الـget الـhas و الـdelete



الـgarbage collection الخاص بالـmemory بمجرد ما يشتغل .. هيحذف الـobjectOne لأنه لم يعد له reference

و طبعا لو استخدمنا الـsize مع الـWeakMap .. النتيجة هتكون undefined

Symbol Iterator And For Of Loop

- الدرس ده هنبدأ ندخل لعالم الـiteration و نعرف ازاي الـiteration بيشتغل

- مبدأيا في الجافاسكريبت عندنا iterable protocols مسئولة عن تنظيم عملية الـiteration على العنصر

- في الجافاسكريبت عندنا data types مجهزة و مصممة للـiteration زي الـarray و الـstring و الـMap و الـSet و بيطلق عليها الـiterators

- عشان العنصر يكون iterable .. لازم يكون بينفذ method اسمها Symbol.iterator

- أي iterator بيكون ليه method اسمها next .. و الـmethod دي بتـ return object فيه 2 properties

الأولى بتمثل الـvalue .. الثانية بتكون boolean value لو كانت true يبقى عملية الـiteration بقت done و لو false يبقى لسه عملية الـiteration مش done

- ممكن نلخص الـiterable و الـiterators في السطور دي

Iterable => Object Has Symbol.Iterator

Iterator

    - Object With Interface Designed For Iteration

    - All Iterators Has Next() Method That Return Result Object

    - The Result Object Has Two Props

    --- [1] Value => The Next Value

    --- [2] Done => Boolean Value To Tell If The Next Value Exists Or No

و نبدأ ناخد أمثلة عشان نفهم

مثال :

let myArray = [1, 2, 3];  // built-in iterable

let myName = "Osama";   // built-in iterable

let myNumber = 1000;

let myObject = {};

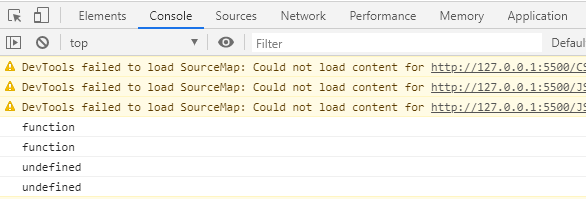
console.log( typeof( myArray[Symbol.iterator] ) );

console.log( typeof( myName[Symbol.iterator] ) );

console.log( typeof( myNumber[Symbol.iterator] ) );

console.log( typeof( myObject[Symbol.iterator] ) );

النتيجة



بما ان الـarray و الـstring من ضمن الـiterators .. النتيجة هتكون function لأنهم مجهزين لعملية الـiteration

- نجرب استخدام الـSymbol.iterator مع الـiterators بالشكل ده

let myArray = [1, 2, 3]; // built-in iterable

let myName = "Osama"; // built-in iterable

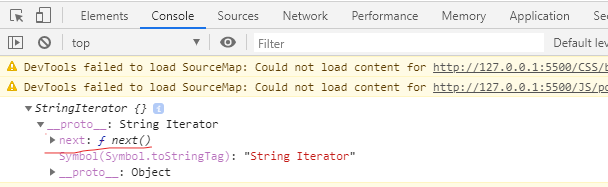
let myNumber = 1000;

let myObject = {};

let theIterator = myName[Symbol.iterator]();

console.log(theIterator);

النتيجة .. هنلاقي object فيه الـnext method



- نجرب استخدام الـ next method بالشكل ده

let theIterator = myName[Symbol.iterator]();

console.log(theIterator);

console.log(theIterator.next());

console.log(theIterator.next());

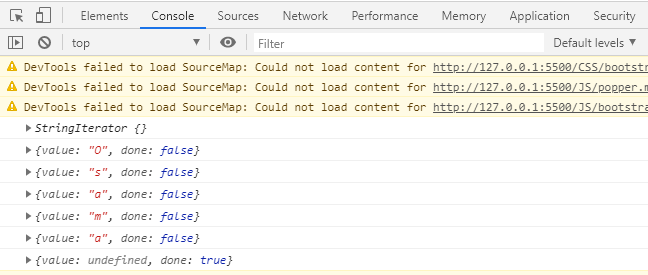
console.log(theIterator.next());

console.log(theIterator.next());

console.log(theIterator.next());

console.log(theIterator.next());

النتيجة



- الـ**for of** loop بيعتمد على الـnext method في عملية الـiteration بالشكل ده

For ... Of Loop

[1] Call Symbol.Iterator With Next Method

[2] Save Value Inside Variable

[3] Check For Done If Its True => Exit The Loop

- نجرب نستخدم الـfor of loop بالشكل ده

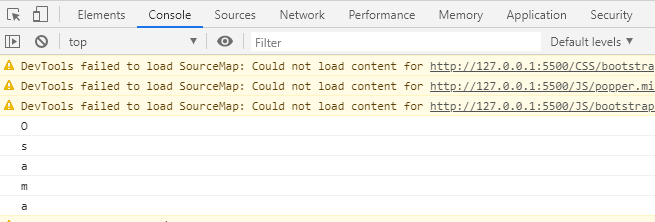
let myName = "Osama";

for (let char of myName) {

    console.log(char);

}

النتيجة



و لو عملنا for of loop على عنصر مش iterable بالشكل ده

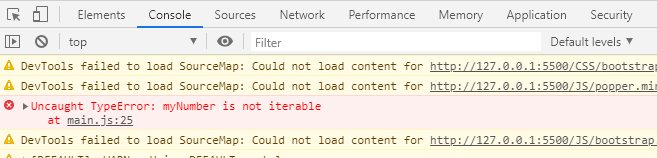
let myNumber = 1000;

for (let num of myNumber) {

    console.log(num);

}

النتيجة



Custom Iterable Object

- الدرس ده هنحول عنصر مش iterable لعنصر iterable

مثال : لو عندنا object عادي و عملنا عليه **for of** loop بالشكل ده

const myObject = {

    name: "Osama",

    age: 36,

    country: "Egypt",

    favSkill: "JavaScript",

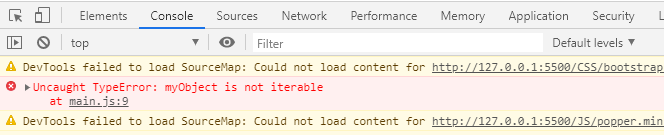
}

for (let prop of myObject) {

    console.log(prop);

}

النتيجة



عشان نحول الـobject لعنصر iterable لازم نستخدم الـsymbol.iterator method بالشكل ده

const myObject = {

    name: "Osama",

    age: 36,

    country: "Egypt",

    favSkill: "JavaScript",

    [Symbol.iterator] () {

        let step = 0;

        let properties = Object.keys(this);

        console.log(properties); // will return ["name", "age", "country", "favSkill"]

        return {

            next() {

                return {

                value: myObject[properties[step]],

                done: step++ === properties.length

                }

            }

        }

    }

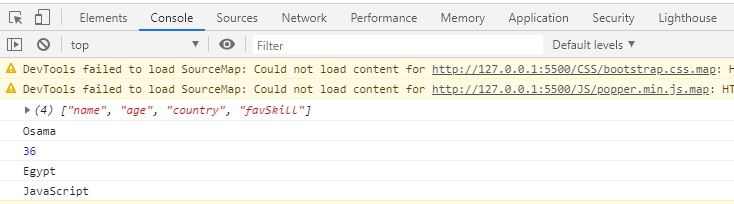
}

for (let prop of myObject) {

    console.log(prop);

}

النتيجة



و بكدا احنا عملنا custom iterator

Generators Part 1 Intro And Whats Generators

- الـgenerator عبارة عن function بتـgenerate قيمة مختلفة في كل مرة ننادي عليها و الـsyntax الخاص بيها شبة الـsyntax بتاع الـfunction العادية باستثناء اننا :

1 – بنحط **\*** بين كلمة function و اسم الـfunction

2- بنستخدم keyword داخل الـ function اسمها **yield**

function \* myGenerator(){

    yield

}

- الـgenerator بتنشئ iterator بشكل تلقائي عشان في كل مرة تنتج value مختلفة

مثال :

function \* generateTickets(){

    yield 1;

    yield 2;

    yield 3;

    yield 4;

    yield 5;

}

let iterator = generateTickets();

console.log( typeof(iterator) );

console.log( iterator.next().value );

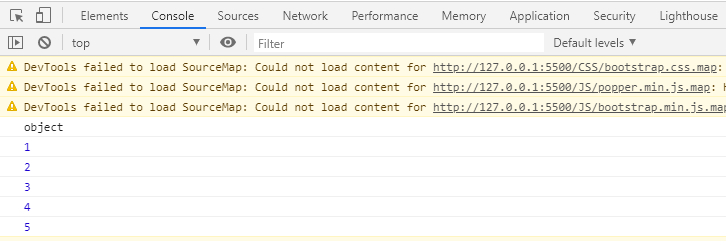
console.log( iterator.next().value );

console.log( iterator.next().value );

console.log( iterator.next().value );

console.log( iterator.next().value );

النتيجة



ملحوظة : لازم تخزن الـgenerator في متغير عشان الـiterator يشتغل و الـgenerator ينتج لك value مختلفة في كل مرة

و إلا الـgenerator هينتجلك فقط أول value في كل مرة تنادي عليه

مثال

function \* generateTickets(){

    yield 1;

    yield 2;

    yield 3;

    yield 4;

    yield 5;

}

console.log( typeof( generateTickets() ) );

console.log( generateTickets().next().value );

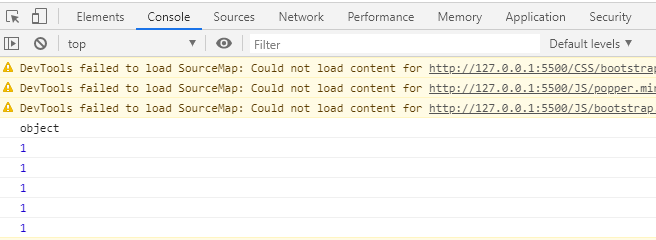
console.log( generateTickets().next().value );

console.log( generateTickets().next().value );

console.log( generateTickets().next().value );

console.log( generateTickets().next().value );

النتيجة



مثال

let myNumbers = [10, 20, 30, 40, 50];

function \* generateNumbers(nums) {

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

        yield nums[i];

    }

}

let generator = generateNumbers(myNumbers);

console.log(generator.next().value);

console.log(generator.next().value);

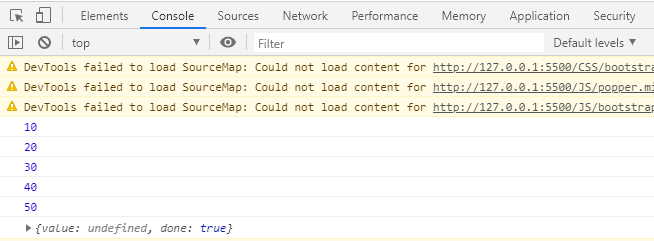
console.log(generator.next().value);

console.log(generator.next().value);

console.log(generator.next().value);

console.log(generator.next());

النتيجة



- بما ان علامة الـ\* بتشير للـgenerator .. نقدر نستخدمها داخل الـgenerator مع الـyield بالشكل ده

function \* generateNums() {

    yield \* [1, 2, 3, 4, 5, 6];

    yield 7;

}

let theGenerator = generateNums();

console.log(theGenerator.next().value);

console.log(theGenerator.next().value);

console.log(theGenerator.next().value);

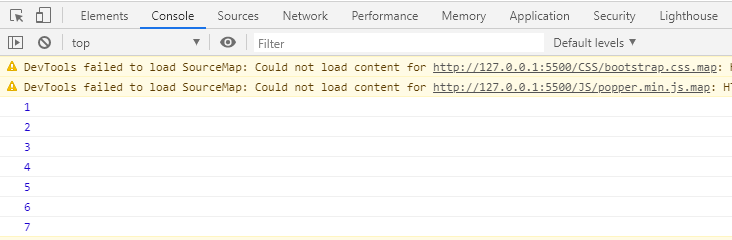
console.log(theGenerator.next().value);

console.log(theGenerator.next().value);

console.log(theGenerator.next().value);

console.log(theGenerator.next().value);

النتيجة



Generators Part 2 Functionality And Return

- بما ان الـgenerator بتـcreate iterator .. اذا نقدر نعمل loop علي الـiterator بالشكل ده

function \* mySkills() {

    yield "HTML";

    yield "CSS";

    yield "PHP";

    yield "Python";

    yield "JavaScript";

}

let theSkills = mySkills(); // Iterator

for (let skill of theSkills) {

    console.log(skill);

}

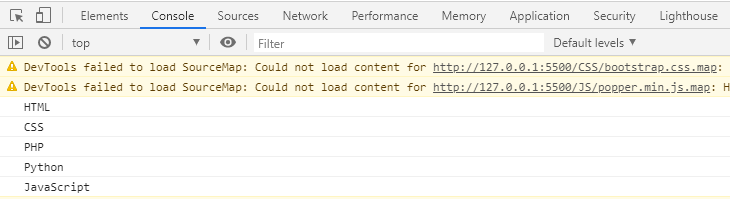
او على الـgenerator function نفسها بالشكل ده

for (let s of mySkills()) {

    console.log(s);

}

و النتيجة مش هتختلف



- الـreturn في الـfunction بتلغي الكود اللي بعدها و بالتالي مش منطقي نستخدم الـreturn في الـgenerator function لأن الغرض من استخدام الـgenerator هو انتاج بيانات بلا توقف

function \* sayNames() {

    yield "Osama";

    yield "Ahmed";

    return "Sayed";

    yield "Eman";

}

let theNames = sayNames();

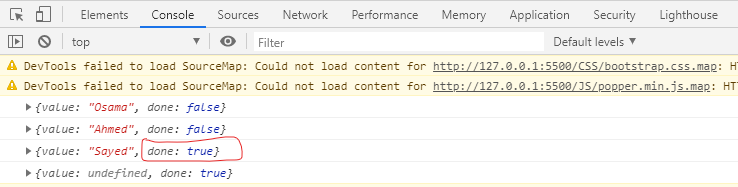
console.log(theNames.next());

console.log(theNames.next());

console.log(theNames.next());

console.log(theNames.next());

النتيجة



نلاحظ ان الـdone بقت true عند Sayed و ده معناه ان الـexecution وقف عندها .. بينما لو كنا استخدمنا yield بدلا من return بالشكل ده

function \* sayNames() {

    yield "Osama";

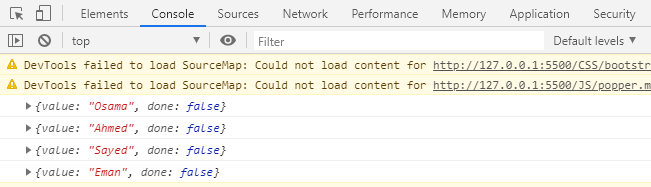
    yield "Ahmed";

    yield "Sayed";

    yield "Eman";

}

الـdone هتكون false بالشكل ده



Generators Part 3 Create Infinite Numbers

- الدرس ده هنعمل مثال على generator بينتج أرقام بشكل لا نهائي بالشكل ده

function \* createInfinite() { // DO NOT LOOP ON THIS

    let i = 0;

    while (true) {

        yield i++;

    }

}

let infiniteIterator = createInfinite();

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

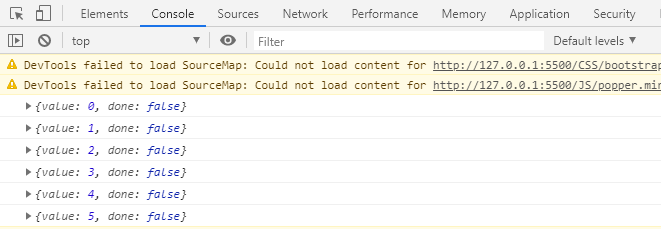
console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

النتيجة



في المثال ده لو كنا عملنا function عادية بدلا من generator function كان هيحصل infinite loop و المتصفح هيهنج .. و هنا بيظهر الفرق بين الـfunction العادية و الـgenerator function و هو اننا بنقدر نتحكم في النتيجة من خلال الـyield

- تطبيقا على نفس المثال .. لو عملنا loop على الـ**iterator** بالشكل ده

function \* createInfinite() { // DO NOT LOOP ON THIS

    let i = 0;

    while (true) {

        yield i++;

    }

}

let infiniteIterator = createInfinite();

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

for (let number of infiniteIterator) {

    if (number > 10) {

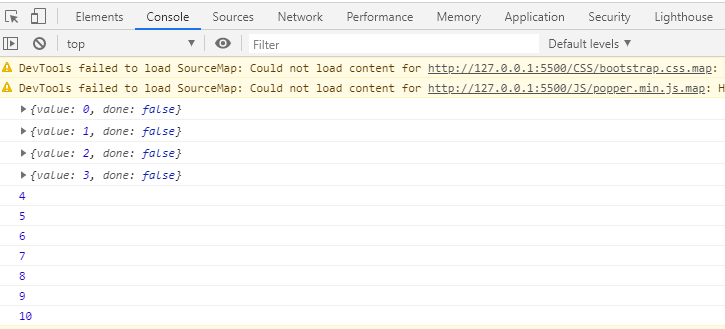
        break;

    }

    console.log(number);

}

الـloop هيبدأ من بعد أخر رقم الـ**iterator** أنتجة بالشكل ده



بينما لو عملنا الـloop على الـ**generator function** نفسها بالشكل ده

let infiniteIterator = createInfinite();

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

console.log(infiniteIterator.next());

for (let number of createInfinite()) {

    if (number > 10) {

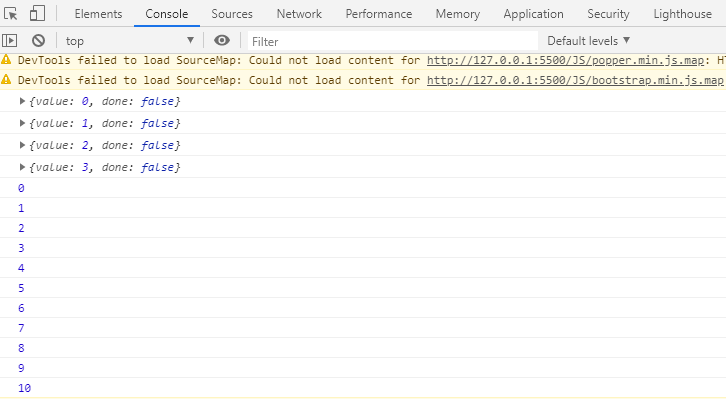
        break;

    }

    console.log(number);

}

الـloop هيبدأ من أول value الـ**generator** **function** هتنتجها بالشكل ده



Generators Part 4 Delegating Generators

- الـdelegating generator هو عبارة عن generator function عادية بنعمل فيها yield لـgenerators functions تانية بالترتيب .. بمعنى انها هتـyield بيانات الـgenerator function الأولى **كلها** ثم الثانية ثم الثالثة و هكذا

مثال

function \* generateNumbers() {

    yield 1;

    yield 2;

    yield 3;

}

function \* generateNames() {

    yield "Osama";

    yield "Ahmed";

    yield "Sayed";

}

function \* generateAll() {

    yield \* generateNumbers();

    yield \* generateNames();

}

let genAll = generateAll();

console.log(genAll.next());

console.log(genAll.next());

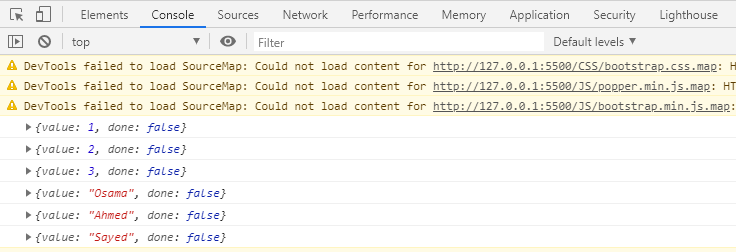
console.log(genAll.next());

console.log(genAll.next());

console.log(genAll.next());

console.log(genAll.next());

النتيجة



New Array Methods Part 1

- عندنا في الـES6 مجموعة array methods جديدة

**Array.of**

- دي method بنقدر من خلالها نـcreate array و الـsyntax بتاعها شكله كدا

Array.of(Elements To Create The Array)

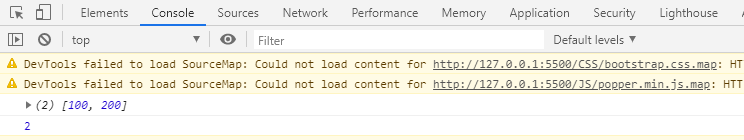
مثال

let newArray = Array.of(100, 200);

console.log(newArray);

console.log(newArray.length);

النتيجة



**Array.from**

- دي method بنقدر من خلالها نعمل array من خلال عناصرiterable أخر و الـsyntax بتاعها شكله كدا

Array.from(Iterable, Map function, This)

مثال

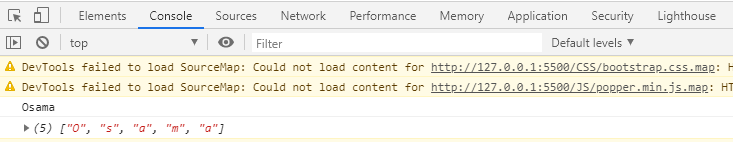
let myName = "Osama";

let arrayOfLetters = Array.from(myName);

console.log(myName);

console.log(arrayOfLetters);

النتيجة



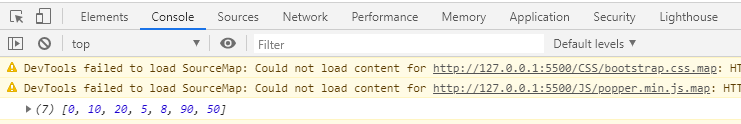
مثال أخر و لكن باضافة Map function على عناصر الـiterable

let myNumbers = [10, 20, 30, 15, 18, 100, 60];

let numbersMinusTen = Array.from(myNumbers, (number) => number - 10);

console.log(numbersMinusTen);

النتيجة



**fill**

- دي method بنقدر من خلالها نملا الـarray بـvalue معينة الـsyntax بتاعها شكله كدا

// arrayName.fill(Value to Fill, Start index - Default 0, End index - Default Array.length)

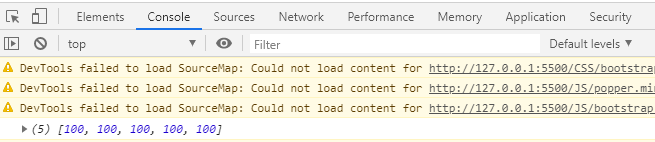
مثال

let theNumbers = [1, 10, 30, 70, 90];

theNumbers.fill(100); // All Will Be 100

console.log(theNumbers);

النتيجة



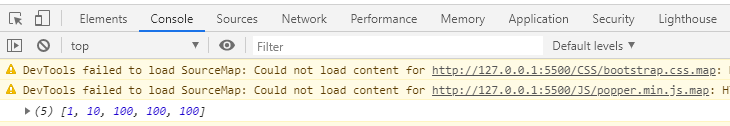
مثال أخر

let theNumbers = [1, 10, 30, 70, 90];

theNumbers.fill(100, 2); // All Will Be 100 starting from 30

console.log(theNumbers);

النتيجة



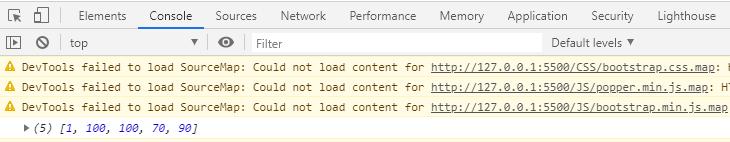
مثال أخر

let theNumbers = [1, 10, 30, 70, 90];

theNumbers.fill(100, 1, 3); // Will replace 10 And 30 with 100

console.log(theNumbers);

النتيجة



New Array Methods Part 2

**find**

- دي method بنقدر من خلالها نبحث في الـarray عنvalue معينة و الـmethod دي زي الـmap بتاخد arrow function بحيث هتنفذها على عناصر الـarray لحد ما تلاقي الـvalue المطلوبة و الـsyntax بتاعها شكله كدا

arrayName.find( arrow function ) Not Returning All Values

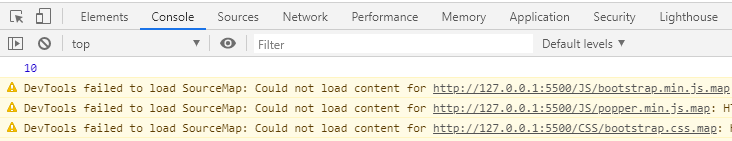
مثال

let myNumbers1 = [1, 10, 20, 30, 40];

let found1 = myNumbers1.find(el => el == 10);

console.log(found1);

النتيجة



**findIndex**

- دي method بنقدر من خلالها نجيب الـindex الخاص بـ value معينة و الـmethod دي زي الـmap بتاخد

arrow function بحيث هتنفذها على عناصر الـarray لحد ما توصل للـindex الخاص بالـvalue المطلوبة و الـsyntax بتاعها شكله كدا

arrayName.findIndex( arrow function ) Not Returning All Values

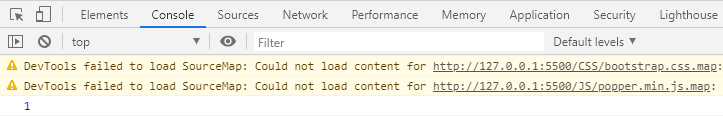
مثال

let myNumbers2 = [1, 10, 20, 30, 40];

let found2 = myNumbers2.findIndex(el => el == 10);

console.log(found2);

النتيجة



**copyWithin**

- دي method بنقدر من خلالها ننسخ عناصر داخل الـarray و نستبدلها بعناصر داخل نفس الـarray و بالتالي عدد عناصر الـarray مش هيزيد و لا هينقص .. الـsyntax شكله كدا

arrayName.copyWithin(Target, Start index - Default 0, End index - Default Array.length)

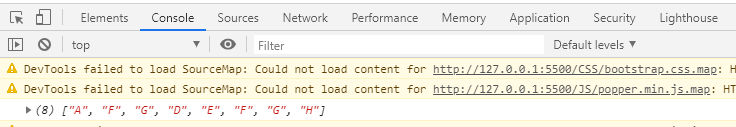
مثال

let myLetters = ["A", "B", "C", "D", "E", "F", "G", "H"];

myLetters.copyWithin(1, 5, 7);

console.log(myLetters);

النتيجة



Modules - Part 1 Import & Export

- الـmodules لا غنى عنها في أي framework و فكرتها قايمة على اننا بنستدعي في ملف الجافاسكريبت بيانات من ملف جافاسكريبت أخر و كل ده بيتم باستخدام **export** keyword و **import** keyword

مثال

عندنا ملف اسمه app.js فيه البيانات دي

const a = 1;

let myArray = [1, 2, 3, 4, 5, 6];

function sayHello(name) {

    return `Hello ${name}`;

}

و عايزين نستخدمها في ملف اسمه main.js .. ده هيتم في 3 خطوات

أولا : في ملف الـHTML هنستدعي الملفين بشكل طبيعي و لكن هنضيفلهم type property و الـvalue هتكون module بالشكل ده

<script src="app.js" type="module"></script>

<script src="main.js" type="module"></script>

ثانيا : في ملف الـapp.js هنعمل export للبيانات اللي محتاجينها في ملف الـmain.js بالشكل ده

export const a = 1;

export let myArray = [1, 2, 3, 4, 5, 6];

export function sayHello(name) {

    return `Hello ${name}`;

}

أو بالشكل ده

const a = 1;

let myArray = [1, 2, 3, 4, 5, 6];

function sayHello(name) {

    return `Hello ${name}`;

}

export{

    a,

    myArray,

    sayHello

}

ثالثا : في ملف الـmain.js هنعمل import للبيانات اللي محتاجينها من ملف الـapp.js بالشكل ده

import {a, myArray, sayHello} from './app.js';

console.log(a);

console.log(myArray);

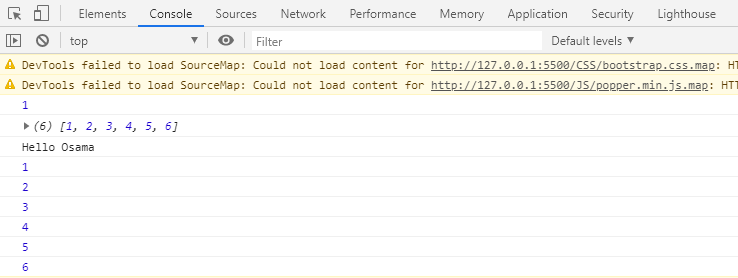
console.log(sayHello("Osama"));

for (let number of myArray) {

    console.log(number);

}

النتيجة



Modules - Part 2 Import All And Aliases

- نقدر و احنا بنعمل import لأي بيانات نديها alias او اسم مستعار من خلال **as** keyword بالشكل ده في ملف الـapp.js

const a = 1;

function sayHello(name) {

    return `Hello ${name}`;

}

export{

    a as myRank,

    sayHello

}

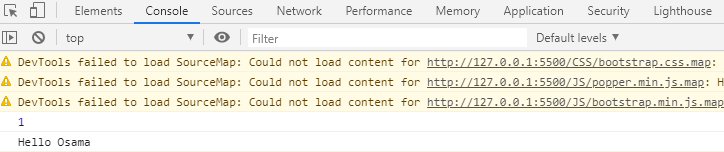
و في ملف الـmain.js هنـexport المتغير **a** بالاسم المستعار بالشكل ده

import {myRank, sayHello} from './app.js';

console.log(myRank);

console.log(sayHello("Osama"));

النتيجة



- ممكن نستدعي كل البيانات اللي في ملف معين بشكل أسهل و أسرع من خلال **import \*** و دي معناها import all

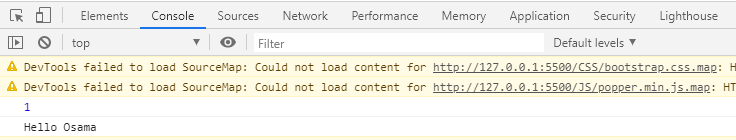
و لازم نعمل alias و ده هيعبر عن object فيه كل بيانات الملف بالشكل ده

import \* as all from './app.js';

console.log(all.myRank);

console.log(all.sayHello("Osama"));

النتيجة



Modules - Part 3 Named Export Vs Default Export

- الـnamed export هو الـexport العادي اللي عملناه الدروس السابقة .. بينما الـdefault export بيسمحلك انك تـexport أي حاجة في ملف الجافاسكريبت و تعملها import في أي ملف أخر بأي إسم .. و كل module بيسمحلك تعمل default export مرة واحدة فقط

مثال

عندنا البيانات دي في ملف الـapp.js

const a = 100;

const b = 200;

export {

    a, b

}

export default function sayHello() {

    return `Hello`;

}

و عايزين نستدعيها في ملف الـmain.js .. فبكل بساطة هنعمل import بالشكل العادي للمتغيرات a و b .. و هنعمل import للـ**sayHello** function بأي إسم بالشكل ده

import { a, b } from './app.js';

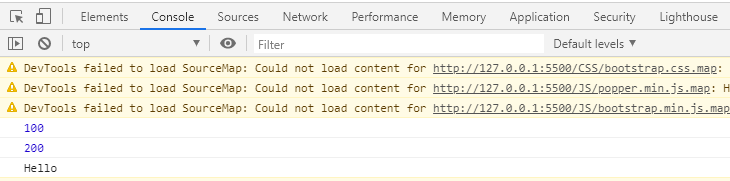
import sayH from './app.js';

console.log(a);

console.log(b);

console.log( sayH() );

النتيجة



- لو عندنا named export و default export .. نقدر نختصر الـimport في سطر واحد بالشكل ده

import sayH , { a, b }  from './app.js';

و لازم نـimport الـdefault export قبل الـnamed export

Classes Part 1 Old and New Syntax

- مبدأ الـOOP في الجافاسكريبت قبل الـES6 كان بيعتمد على الـConstructor function .. و الـConstructor function عبارة عن function عادية بننشئ بيها objects ليها نفس الـproperties و لكن بـvalues مختلفة

مثال توضيحي :

تخيل انك بتعمل system لمصنع موبايلات .. و كل موبايل هيتعمله object هيتسجل فيه بياناته زي الـserial number و الـcolor و الـprice

في البرمجة التقليدية .. هتلاقي نفسك بتعمل object لكل موبايل و تسجل فيه الـvalues بشكل يدوي بالشكل ده

const phone1 = {

    serial: 123,

    color: "Red",

    price: 500,

};

const phone2 = {

    serial: 159,

    color: "Black",

    price: 500,

};

const phone3 = {

    serial: 167,

    color: "Silver",

    price: 500,

};

و طبعا ده أسلوب غير منطقي و مرهق .. لأن لو احتاجت تعمل تعديل على كل الموبايلات هتلاقي نفسك مضطر تعدي على كل object و تعدل فيه و من هنا ظهر فكرة الـconstructor

- جرى العرف ان اول حرف من اسم الـconstructor يكون capital .. فمثلا هنعمل constructor function هنسميها User و هنديها parameters هتمثل الـvalues اللي هتتسجل في الـproperties .. و داخل الـfunction هنعمل declaration للـproperties اللي هتكون في الـobject باستخدام الـ**this** keyword اللي بدورها هتشير لأي object هيتعمل بالشكل ده

function User(theName, theAge) {

    this.name = theName;

    this.age = theAge;

}

لو حبينا نضيف methods للـConstructor function من خارج الـConstructor function يبقى هنضيفها من خلال

الـprototype property و دي عبارة عن built in property في أي function بنعملها في الجافاسكريبت بما في ذلك الـconstructor function

بالشكل ده

// adding methods to the User Constructor Function

User.prototype.getInfo = function(){

    console.log(`Hello ${this.name} Your Age Is ${this.age}`);

}

User.prototype.getDays = function(){

    console.log(`You Lived For ${this.age \* 365} Days`);

}

لو احنا عايزين نعمل object .. فبكل بساطة هنعمل كدا

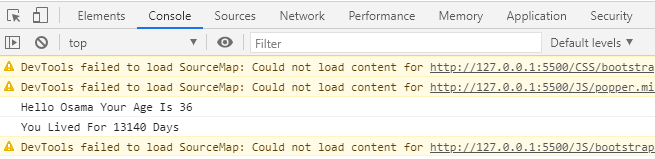
// create objects

const firstUser = new User("Osama", 36);

firstUser.getInfo();

firstUser.getDays();

النتيجة



- بداية من الـES6 .. تم اضافة الـclass و أصبح ممكن نستخدم الـclass في الجافاسكريبت بدلا من الـconstructor function و نعمل منها objects زي باقي لغات البرمجة

نفس الـUser constructor function هتبقى كدا بالـclass شكلها كدا

class User {

    constructor(theName, theAge) {

        // "this" represents The Created Object From The Constrcutor Function

        this.name = theName;

        this.age = theAge;

    }

    getInfo() {

        console.log(`Hello ${this.name} Your Age Is ${this.age}`);

    }

    getDays() {

        console.log(`You Lived For ${this.age \* 365} Days`);

    }

}

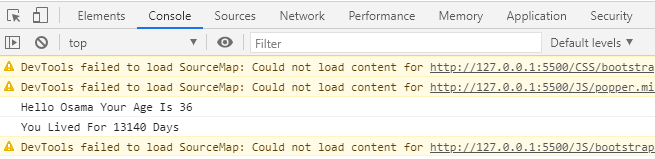
و بنعمل objects من الـclass بنفس الطريقة بالشكل ده

const firstUser = new User("Osama", 36);

firstUser.getInfo();

firstUser.getDays();

و النتيجة مش هتتغير



Classes Part 2 Extend Class From Another Module

- الـinheritance معناه ان class تورث كل الـproperties و الـmethods من class تانية

مثال : تخيل فيه class اسمه User في ملف عندنا اسمه app.js بالشكل ده

class User {

    constructor(name, email) {

        this.name = name;

        this.email = email;

    }

    getInfo() {

        console.log(`Hello ${this.name} Your Email is ${this.email}`);

    }

}

و عملنا default export ليه بالشكل ده

export default User;

لو عايزين مثلا في ملف main.js نعمل class اسمه Admin هيكون فيه نفس الـproperties و الـmethods بتاعة

الـUser class .. يبقى أولا هنعمل import للـUser class من ملف app.js بالشكل ده

import User from './app.js';

ثانيا هنعمل inheritance للـAdmin class من خلال الـextends keyword بالشكل ده

class Admin extends User {

    constructor(name, email) {

        super(name, email);

    }

}

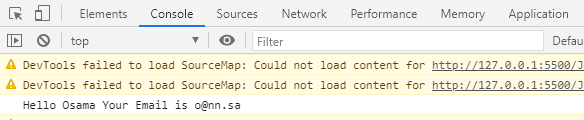
معلومة : داخل الـ child constructor بنستدعي function اسمها super() و دي بتـcall الـparent constructor و بنديها نفس الـparameters اللي في الـparent constructor

نجرب نعمل object بالشكل ده

let firstAdmin = new Admin("Osama", "o@nn.sa");

firstAdmin.getInfo();

النتيجة



Call Stack, Web API, Event Loop, Callback Queue

- الدرس ده هنتكلم عن مجموعة مفاهيم قبل ما ندخل في عالم الـPromise

**1- Call Stack :** هو ألية أو mechanism بيستخدمها مترجم لغة الجافاسكريبت عشان يعمل Tracking للكود بتاعك .. فبمجرد ما تنادي على function .. مترجم اللغة بيضفها داخل الـStack .. و بعد ما الـfunction بتتنفذ .. مترجم اللغة بيحذفها من الـStack

- Call Stack

--- Mechanism To Make Interpreter Track Your Calls

--- When You Call A Function Its Added To The Stack

--- When Function Executed Its Removed From The Stack

--- The Interpreter Continue Calling From The Last Point Reached

--- Call Stack Detect Web API And Leave It To The Browser To Handle It

مثال

function first() {

    console.log("Hello First");

}

function second() {

    first();

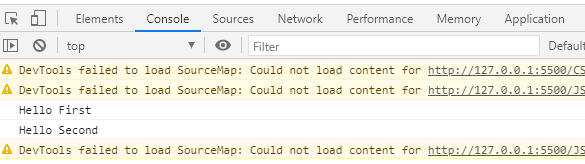
    console.log("Hello Second");

}

second();

اللي هيحصل هنا .. مترجم اللغة هيضيف الـsecond function داخل الـStack عشان ينفذها .. تنفيذ الـsecond function بيتضمن تنفيذ function تانية اسمها first .. فيقوم مترجم اللغة باضافة الـfirst function داخل الـStack عشان ينفذها .. و بعد تنفيذ الـfirst function يقوم مترجم اللغة بحذفها و يرجع يكمل تنفيذ الـsecond function و بعد ما يخلص تنفيذها يحذفها من الـStack

النتيجة



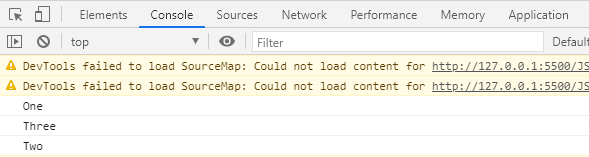
كلام منطقي مفيش فيه أي مشكلة أو جديد .. لكن لو عملنا مثال تاني بالشكل ده

console.log("One");

window.setTimeout(() => console.log("Two"), 0);  // Window in JavaScript refers to the browser

console.log("Three");

النتيجة هتكون



تفسير اللي حصل هو ان الـCall Stack لما بيكتشف أي method من الـ Web API methods بيسيبها للـbrowser ينفذها بنفسه بدلا من مترجم اللغة .. و الـsetTimeout من ضمن الـWeb API methods

**2- Web API :** هو مجموعة methods خاصة بالـbrowser .. و الـmethods دي لما بتتنفذ .. الـcallbacks او النتيجة بتاعتها بتتخزن في الـCallback Queue

- Web API

--- Methods Available From Environment => Browser

--- When Complete It Add The Callback To The Callback Queue

**3- Callback Queue :** هو مجموعة الـCallbacks اللي راجعة من الـWeb API methods

**4- Event Loop :** عبارة عن أليه أو mechanism بيقوم بالأتي

1- بيشيك على تنفيذ كل ما بداخل الـCall Stack

2- بياخد كل الـCallbacks من الـCallback Queue و يضيفها للـCall Stack عشان تتنفذ

- Event Loop

--- Wait The Call Stack To Finish

--- Get Callback From Callback Queue

--- Add Callback To Call Stack

مثال أخر

window.setTimeout(() => console.log(myVariable), 0);

let myVariable = 100;

myVariable += 200;

النتيجة 300

اللي حصل كالأتي

1- الـCall Stack هيمشي سطر بسطر .. هيتجاهل الـsetTimeout لأنها Web API method و هينفذ الاكواد اللي بعدها

2- الـbrowser هينفذ الـsetTimeout و هيخزن الـCallback في الـCallback Queue

3- الـEvent Loop هياخد الـCallback اللي في الـCallback Queue و يتأكد ان الـCall Stack خلص شغله .. فيضيفها للـCall Stack عشان ينفذها

مثال أخر

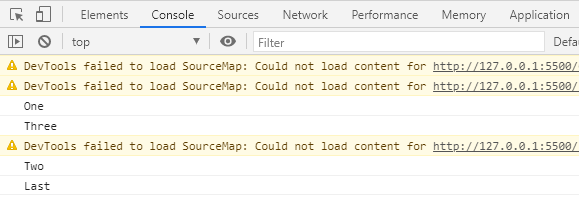
window.setTimeout(() => console.log("Last"), 1000);

console.log("One");

window.setTimeout(() => console.log("Two"), 0);

console.log("Three");

النتيجة



Asynchronous vs Synchronous Programming

- الدرس ده هنتكلم عن الفرق الـSynchronous Programming و الـAsynchronous Programming

- مبدأيا الجافاسكريبت تعتبر single threaded language يعني فيها One Call Stack و ده معناه ان كل task بتتنفذ لوحدها و مينفعش أكتر من task يتنفذوا في وقت واحد .. بينما فيه لغاتmulti-threaded language زي C++ و Java ممكن أكتر من task يتنفذوا في وقت واحد

- الـSynchronous Programming معناها البرمجة المتزامنة و فيها كل task لازم تتنفذ عشان اللي بعدها تتنفذ

مثال بسيط

console.log("One");

window.alert("Bad");

console.log("Two");

بما ان الـalert بيعطل كل حاجة .. اذا الـtask اللي بعد الـalert مش هتتنفذ الا لما الـalert يخلص

- الـAsynchronous Programming معناها البرمجة الغير متزامنة و فيها ممكن task تتنفذ بدون انتظار تنفيذ tasks أخرى

مثال بسيط من الواقع للتوضيح

في الفيسبوك .. لما بتدخل على group و تعمل scroll عشان تستعرض الـposts .. في الجانب الأيمن من الصفحة بتلاقي box بيحصل فيه loading للاصدقاء عشان لو عايز تعمل invite لأي صديق .. عملية الـloading دي بتتنفذ بشكل Asynchronous لأنها بتتنفذ مع عملية الـscrolling في وقت واحد و مش مرتبطة بأي actions في الصفحة

و هي دي فكرة الـAsynchronous Programming

Promise Intro And Discussions

- الـPromise من أهم الإضافات في الـES6 و بتسهل الـAsynchronous Programming .. و كلمة Promise تعني وعد و الـPromise في الجافاسكريبت بالفعل عبارة عن وعد بالحصول على value في المستقبل

Promise Intro

- Promise Is One Of The Best Features In ES6

- Promise Make It Easy To Support Asynchronous Programming

- Promise in JavaScript Is Like Promise in Real Life

- Promise Is Something That Will Happen in The Future

- Promise Is A Placeholder For A Future Value

- You Booked A Cinema Ticket, You Have A Promise From Cinema Owner To Have A Seat

- You Got Task To Do, U Promised Your Boss To Do The Task

- You Ordered Food From Restaurant

--- [ Promise ] Is The Order Record

--- [ The Future Value ] Is The Food You Will Receive

- You Ordered iPhone From Souq Shop

--- [ Promise ] Is The Order Record

--- [ The Future Value ] Is The iPhone You Will Receive

- الـPromise عبارة عن object بيمثل نتيجة الـAsynchronous operation

- الـPromise بيستخدم الـCallbacks في الـAsynchronous Programming و بيتجنب مشكلة الـpyramid of DOM او مايعرف بالـCallback Hell و دي مشكلة بتحصل لما لما function تنادي على function بتنادي على function بتنادي على function .. هنشوف مثال على المشكلة دي في الدروس القادمة

- JavaScript Use Callbacks For Asynchronous Programming

--- [1] Function Called And Do The Task

--- [2] Action Complete

--- [3] Another Function Called

--- [4] Action Complete

--- [5] Another Function Called

- In Complex Cases => Every Call Add Nesting Level

- <http://callbackhell.com/>

- Promise Avoiding Callback Hell Or Pyramid Of Doom

- حالات الـPromise اما

**pending** و دي معناها ان الـoperation لم يتنهي

**fulfilled** و دي معناها ان الـoperation نجح

**rejected** و دي معناها ان الـoperation نجح

[1] Pending => Not Fulfilled Or Rejected

[2] Fulfilled => Operation Succeeded

[3] Rejected => Operation Failed

 Create Your First Promise

- الـsyntax بتاع الـPromise object شكله كدا

const varName = new Promise( executer function with 2 params )

مثال

const myPromise = new Promise( (resolve, reject) => {

    /\*

        Asynchronous Operation

        Fulfilled => resolve

        Rejected => reject

    \*/

} )

- الـresolve و الـreject بيمثلواhandlers للـPromise و كل واحدة فيهم عبارة عن function .. فالـPromise أول ما بيبدأ بيكون pending .. و لو الـPromise تم بنجاح .. الـresolve function هتتنفذ .. أما لو الـPromise فشل ..

الـreject function هتتنفذ

مثال من الحياة العملية

هنفترض اننا هنعملobject Promise عشان نـconnect بالـdatabase .. فلو الـconnection تم بنجاح هننفذ الـresolve function و لو فشل هننفذ الـreject function بالشكل ده

const myPromise = new Promise( (resolve, reject) => {

    let connect = true; // connection status

    if (connect) {

        resolve("Connection Established") ;

    } else {

        reject( Error("Connection Failed") );

    };

} );

احنا كدا عملنا الـimplementation بتاع الـPromise object .. عشان نشغل الـPromise بنستخدم method في الـPromise object اسمها then بتاخد 2 parameters

الـ parameter الأول عبارة عن function بتمثل الـresolve function

الـ parameter التاني عبارة عن function بتمثل الـreject function

بالشكل ده

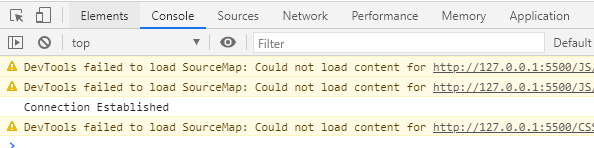
myPromise.then(

    (resolved) => console.log(resolved), // represents resolve

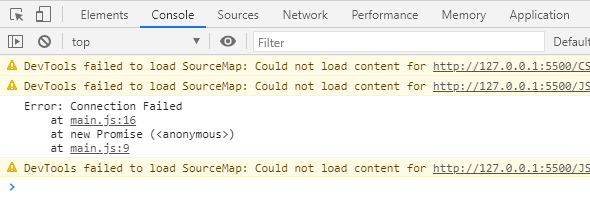
    (rejected) => console.log(rejected)  // represents reject

)

النتيجة



و لو غيرنا قيمة الـconnect لـfalse .. النتيجة هتكون



تحسبا للحالات اللي هيكون فيها كود طويل في الـthen .. من باب تبسيط الكود ممكن نعمل كدا

const theResolved = (resolved) => console.log(resolved);

const theRejected = (rejected) => console.log(rejected);

myPromise.then( theResolved, theRejected )

و النتيجة مش هتختلف

- ممكن نعمل chaining للـPromise object بالشكل ده

const myPromise = new Promise( (resolve, reject) => {

    let connect = true; // connection status

    if (connect) {

        resolve("Connection Established") ;

    } else {

        reject( Error("Connection Failed") );

    };

} ).then(

    (resolved) => console.log(resolved),  // represents resolve

    (rejected) => console.log(rejected)   // represents reject

)

Then And Catch And Training

- الدرس ده هناخد معلومات بسيطة

- الـPromise object بيتنفذ فيه state واحدة فقط .. اما الـresolve و اما الـreject .. مثال

const myPromise = new Promise( (resolve, reject) => {

        // Only One State Allowed

        resolve("Resolved");

        reject("Rejected");

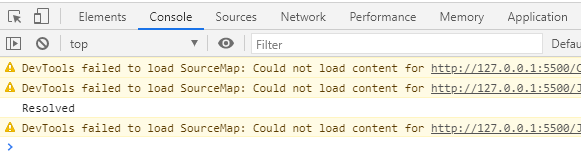
} )

const theResolved = (resolved) => console.log(resolved);

const theRejected = (rejected) => console.log(rejected);

myPromise.then( theResolved, theRejected )

النتيجة



- الـthen في الـPromise بتـhandle الـresolve و الـreject .. بينما عندنا method في الـPromise اسمها catch بتـhandle الـreject فقط

مثال

const myPromise = new Promise( (resolve, reject) => {

    let connect = false; // connection status

    if (connect) {

        resolve("Connection Established") ;

    } else {

        reject( Error("Connection Failed") );

    };

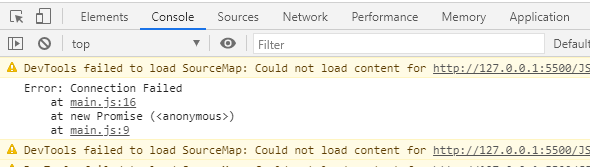
} )

const theResolved = (resolved) => console.log(resolved);

const theRejected = (rejected) => console.log(rejected);

myPromise.catch(theRejected)

النتيجة



و نقدر نستخدم الـthen و الـcatch مع بعض .. بحيث الـthen تـhandle الـresolve و الـcatch تـhandle الـreject بالشكل ده

myPromise.then(theResolved).catch(theRejected)

و نقدر من خلال الـthen نـhandle الـreject فقط .. من خلال وضع null مكان الـresolve بالشكل ده

const myPromise = new Promise( (resolve, reject) => {

    let connect = true; // connection status

    if (connect) {

        resolve("Connection Established") ;

    } else {

        reject( Error("Connection Failed") );

    };

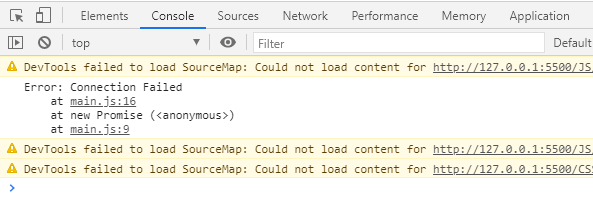
} )

const theResolved = (resolved) => console.log(resolved);

const theRejected = (rejected) => console.log(rejected);

myPromise.then(null, theRejected);

النتيجة .. الـresolve مش هيحصله handle و الـconsole هيبقى فاضي .. بينما لو الـconnection كان false النتيجة هتكون



لأننا عاملين handle للـreject

- الـPromise بيحل مشكلة الـCallback hell .. لأننا في الـPromise بنعمل attach للـfunction و ليس pass

مثال : عندناcase Callback مشهورة على الانترنت بالشكل ده

firstRequest(function (response) {

    secondRequest(

        response,

        function (nextResponse) {

            thirdRequest(

                nextResponse,

                function (finalResponse) {

                    console.log("Final response: " + finalResponse);

                },

                failureCallback

            );

        },

        failureCallback

    );

}, failureCallback);

لو جينا نستخدم الـPromise مع الـcase دي .. شكل الكود هيكون كدا

firstRequest()

.then((response) => secondRequest(response))

.then((nextResponse) => thirdRequest(nextResponse))

.then((finalResponse) => console.log("Final response: " + finalResponse))

.catch(failureCallback);

نلاحظ ان الكود بقى أوضح و أسهل

Promise And XMLHTTPRequest

- بعد ما اتعلمنا الـAJAX في القناة .. الدرس ده هنستخدم الـPromise في اننا نعمل AJAX Request عشان نجيب بيانات من API

- مبدأيا .. هنعمل AJAX Request لـGitHub Repos API عشان نجيب الـRepos الخاصة بـuser اسمه ElzeroWebSchool بالشكل ده

let myRequest = new XMLHttpRequest();

// XMLHttpRequest.open(Method, URL, Async, User, Pass)

myRequest.open("GET", "https://api.github.com/users/ElzeroWebSchool/repos", true);

// Send The Request

myRequest.send();

myRequest.onreadystatechange = function () {

    if (this.readyState === 4 && this.status === 200) {

    /\*

        Ready State => The Status Of The Request

        [0] Request Not Initialized

        [1] Server Connection Established

        [2] Request Received

        [3] Processing Request

        [4] Request Is Finished And Response Is Ready

        Status => Response Status Code

        [200] Ok

        [404] Not Found

    \*/

    console.log(JSON.parse(this.responseText));

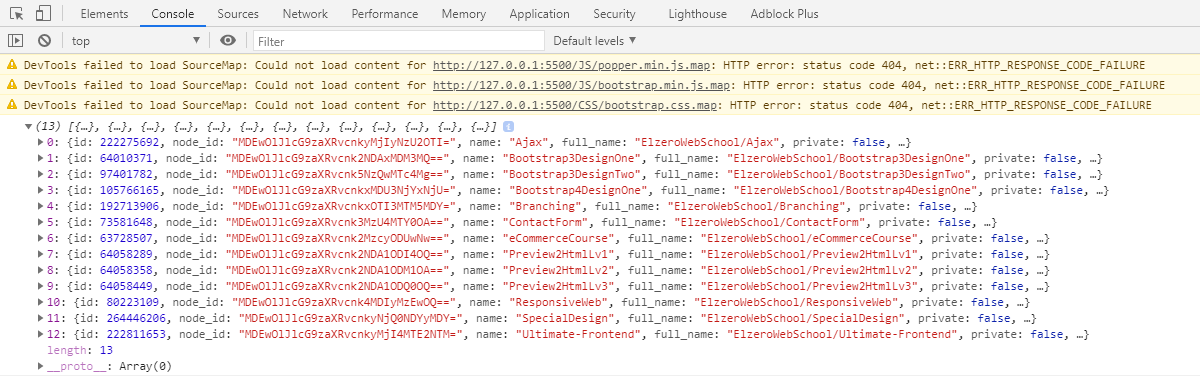
    } else{

            console.log(this.statusText)

        }

};

لو الـrequest تم بنجاح .. هنلاقي الـRepos اتعرضت في الـconsole بالشكل ده



هنعمل تحسينات على الكود باننا هنخلي الـRequest داخل function هنسميها getFirstRepo و هنعمل parameter هيمثل الـAPI URL عشان الـfunction تكون dynamic .. و هنجيب الـname بتاع أول Repo فقط بدلا من كل الـRepos بالشكل ده

function getFirstRepo(apiURL) {

    let myRequest = new XMLHttpRequest();

    myRequest.open("GET", apiURL, true);

    myRequest.send();

    myRequest.onreadystatechange = function () {

        if (this.readyState === 4 && this.status === 200) {

            console.log( JSON.parse(this.responseText)[0].name );

        }else{

            console.log(this.statusText)

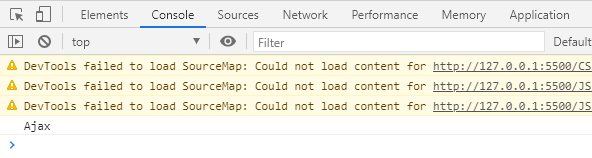
        }

    };

}

getFirstRepo("https://api.github.com/users/ElzeroWebSchool/repos")

النتيجة



هنعمل نفس الـRequest و لكن باستخدام الـPromise بالشكل ده

const getFirstRepo = (apiURL) => {

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

        let myRequest = new XMLHttpRequest();

        myRequest.open("GET", apiURL, true);

        myRequest.send();

        myRequest.onload = function () {

            if (this.readyState === 4 && this.status === 200) {

                resolve( JSON.parse(this.responseText)[0].name );

            }else{

                reject(Error( this.statusText ));

            };

        };

    } );

}

ملاحظات على الكود :

1- الـ onreadychange مش بتشتغل مع الـ Promise فبنستخدم method اسمها onload بدلا من onreadychange

2- هنـcall الـresolve لو الـRequest نجح .. ولو فشل هنـcall الـreject

و هنـcall الـfunction و هنمررلها API URL .. و بما ان الـfunction بتـreturn Promise .. اذا هنستخدم الـthen عشان ننفذ الـPromise بالشكل ده

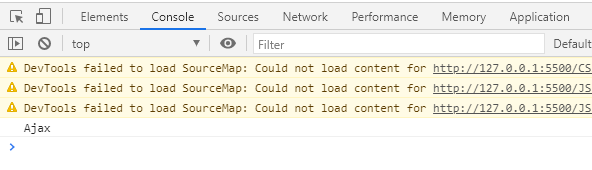
getFirstRepo("https://api.github.com/users/ElzeroWebSchool/repos").then(

    (result) => console.log(result),

    (error) => console.log(error)

);

النتيجة



Ignoring The Pyramid Of Doom

- الدرس ده هنعمل مثال عن ازاي نتفادى الـ Callback hell من خلال الـPromise و ازاي نعمل chain للـthen methods

- هنفترض اننا هنعمل Promise object بيـaccess كل الـposts في الداتابيز بالشكل ده

const myPromise = new Promise((resolve, reject) => {

    let thePosts = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"];

    resolve(thePosts);

  });

لو عايزين نعرض مثلا عدد الـposts .. وأول post .. وأخر post .. وعدد الصفحات باعتبار ان كل صفحة هيكون فيها 2 posts

فبكل بساطة هنعمل chain of then methods .. بس كل then لازم تـreturn الـposts array عشان الـthen اللي بعدها تستلمه و تشتغل عليه بالشكل ده

myPromise

.then((result) => {

    console.log(`Total Number Of Posts Is ${result.length} Posts`);

    return result;

})

.then((result) => {

    console.log("#".repeat(15));

    console.log(`The First Post Is ${result[0]}`);

    return result;

})

.then((result) => {

    console.log("#".repeat(15));

    console.log(`The Last Post Is ${result[result.length - 1]}`);

    return result;

})

.then((result) => {

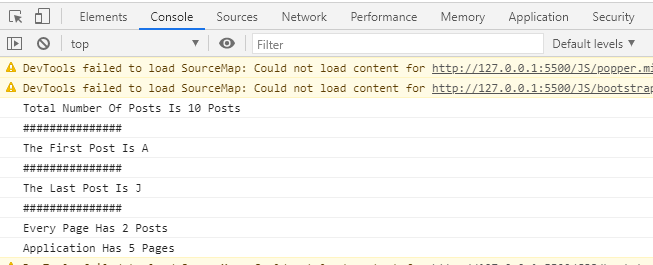
    console.log("#".repeat(15));

    console.log(`Every Page Has 2 Posts`);

    console.log(`Application Has ${result.length / 2} Pages`);

});

النتيجة



الكود أبسط بكتير من من عمل الـCallback Hell

Promise - Fetch And Training

- الدرس ده هناخد فكرة بسيطة عن الـfetch و متستقبلا هيكون فيه playlist مخصصة للـfetch

- الـfetch عبارة عن built-in method مستقله بتاخد parameter عبارة عن الـAPI URL و بتـreturn Promise

- بما ان الـfetch بتـreturn Promise يبقى هنستخدم معاها الـthen method عشان تتنفذ

مثال

بالتطبيق على نفس فكرة مثال الدرس السابق و لكن الاختلاف اننا هنـaccess الـRepos API عشان نجيب كل الـRepos بالشكل ده

fetch("https://api.github.com/users/ElzeroWebSchool/repos")

.then((result) => {

    let allRepos = result.json();

    console.log(allRepos);

})

النتيجة



لو عايزين مثلا نعرض بيانات زي عدد الـRepos و اسم أول Repo و اسم أخر Repo

فبكل بساطة هنعمل chain of then methods .. بس كل then لازم تـreturn الـRepos عشان الـthen اللي بعدها تستلمها و تشتغل عليها بالشكل ده

fetch("https://api.github.com/users/ElzeroWebSchool/repos")

.then((result) => {

    let allRepos = result.json();

    console.log(allRepos);

    return allRepos;

})

.then((repos) => {

    console.log(`Total Number Of Repos Is ${repos.length}`);

    return repos;

})

.then((repos) => {

    console.log(`First Repo Is ${repos[0].name}`);

    return repos;

})

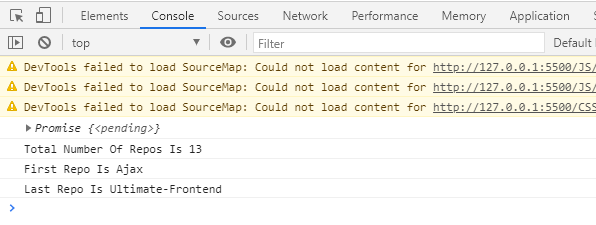
.then((repos) => {

    console.log(`Last Repo Is ${repos[repos.length - 1].name}`);

    return repos;

});

النتيجة



Promise - All And Race

**Promise.all()**

- لو شغالين في project وعندنا أكتر من Promise و عايزين كل الـPromises ينجحوا لتطبيق فكرة ما .. وقتها بنستخدم method اسمها Promise.all( ) و دي بتاخد parameter عبارة عن array فيها الـPromises اللي عايزينهم ينجحوا كلهم

مثال .. عندنا 2 Promises بالشكل ده

const myFirstPromise = new Promise((resolve, reject) => {

    let connect = true;

    if (connect) {

        resolve("First Promise Resolved");

    } else {

        reject("First Promise Rejected");

    }

});

const mySecondPromise = new Promise((resolve, reject) => {

    let settings = true;

    if (settings) {

        resolve("Second Promise Resolved");

    } else {

        reject("Second Promise Rejected");

    }

});

و عايزين لو حصل resolve في الـ2 Promises .. هنطبع في الـconsole عبارة All Promises Done

فبكل بساطة هنستخدم الـPromise.all( ) بالشكل ده

Promise.all( [myFirstPromise, mySecondPromise] ).then( (result) => {

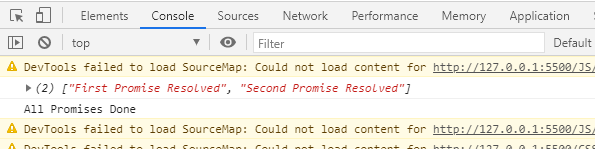
    console.log(result);

    console.log("All Promises Done");

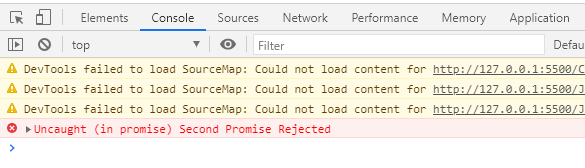
} );

بما ان الـconnect في الـmyFirstPromise بـtrue و الـsettings في الـmySecondPromise بـtrue

اذا الـ2 promises هيحصلهم resolve و النتيجة هتكون



بينما لو الـsettings كانت false .. النتيجة هتكون



**Promise.race()**

- لو شغالين في project وعندنا أكتر من Promise بيتعاملو مع نفس الداتا .. و عايزين ننفذ أول Promise يعمل resolve بغرض تطبيق فكرة ما .. وقتها بنستخدم method اسمها Promise.race( ) و دي بتاخد parameter عبارة

عن array of Promises

مثال .. عندنا 2 Promises بالشكل ده

const myFirstPromise = new Promise((resolve, reject) => {

    window.setTimeout(() => {

    resolve("First Promise Resolved");

    }, 2000);

});

const mySecondPromise = new Promise((resolve, reject) => {

    window.setTimeout(() => {

    resolve("Second Promise Resolved");

    }, 1000);

});

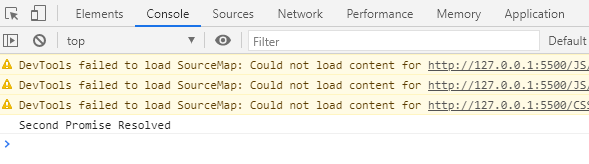
و عايزين ننفذ الـPromise اللي هينجح الأول .. فهنستخدم الـPromise.race( ) بالشكل ده

Promise.race([myFirstPromise, mySecondPromise]).then((result) => {

    console.log(result);

});

بما ان الـmySecondPromise هيعمل resolve أولا لأنه هيستغرق وقت مدته ثانية .. اذاالنتيجة هتكون



The End And References

- لا يوجد أي Code في الدرس و الدرس كله مراجع و مصادر للتعمق