15 – iteration

* مفهوم التكرار مهم جدا انك تفهمه في جافا اسكريبت وتفهم ازاي بيحصل التكرار وليه في حاجات بيتم عليه لوب وحاجات بترفض ده

EX:

```
let num = 10;
let str = "ahmed";
let arr = [100, 200, 300];
let obj = { name: "ahmed", age: 10 };

for (let x of num) {
    console.log(x);
} // numbers isn't iterable
for (let x of obj) {
    console.log(x);
} // objects isn't iterable
for (let x of str) {
    console.log(x);
} // strings is iterable
for (let x of arr) {
    console.log(x);
} // arrays is iterable
```

* المثال بيوضح ان في أنواع بتقبل loop وأنواع لا
* تعالوا بقي نفهم الأنواع اللي بتقبل بتقبل ليه واللي بترفض بترفض ليه
* نبدأ باللي بيقبل زي array and strings
* الفكره كلها ان لما بنشأ متغير بيحمل قيمه نوع جديد من أنواع البيانات.. المتغير ده بيورث من الاوبجيكت الأساسي بتاعه methods and properties اللي موجوده فيه ولذلك انت لو مثلا عندك string بتطبق عليه أي method عادي .. طب تعالوا نبص بصه علي ال object الأساسي بتاع أي نوع

EX:

```
console.log(String.prototype)
console.log(Number.prototype)
```

* هنا هيظهر الاوبجيكت وجواه كل حاجه خاصه بالنوع ده نقدر نستخدمها .. تحت خالص هتلاقي عندك في string حاجه اسمها Symbol(Symbol.iterator) ودي بقي الخاصه أصلا بعمليه loop ولذلك مش هتلاقيها في number or object

* تعالوا بقي نتأكد من الكلام ده علي المتغيرات بتاعتنا

EX:

```
var num = 10;
var str = "ahmed";
var arr = [100,200,300];
var obj = {name:"ahmed",age:10};
let num = 10;
let str = "ahmed";
let arr = [100, 200, 300];
let obj = { name: "ahmed", age: 10 };
console.log(num[Symbol.iterator]); //undefined
console.log(obj[Symbol.iterator]); //undefined
console.log(arr[Symbol.iterator]); //function
console.log(str[Symbol.iterator]); //function
```

* بما انها method تعالوا نشغلها ونشوف هيطلع ليا ايه

EX:

```
console.log(arr[Symbol.iterator]()); //function
console.log(str[Symbol.iterator]()); //function
```

- لو دخلنا جواه هنلاقي method اسمها next() طب تعالوا كده نجرب نستخدمها ونشوف هيطلع ايه

EX:

```
const arr = [100, 200, 300];
let iterator = arr[Symbol.iterator]();
console.log(iterator.next()); //v = 100 ; d = false
console.log(iterator.next()); //v = 200 ; d = false
console.log(iterator.next()); //v = 300 ; d = false
console.log(iterator.next()); //v = undefined ; d = true
```

* هتلاقي الناتج عباره عن object جواه 2 props الاولي value والتانيه done
* يعني كل مره بيروح يجيب العنصر اللي بعده ويديك ok انه خلاص done ودي فكره عمل ال loop

16- generator function

* تقدر تشغلها علي مراحل داخلها بعكس regular function اللي بتنفذ الكود داخلها مره واحده

syntax :

Funnction * name(){

Yield value ;

}

EX:

```
function* gen() {
    yield console.log("AHMED");
    yield console.log("yasser");
    yield console.log("belal");
}
let fun = gen();
console.log(fun.next());
console.log(fun.next());
console.log(fun.next());
```

* you can use array

EX1:

```
let numbers = [1, 2, 3, 4, 5, 6]
function* gen(x) {
    for (let i = 0; i < x.length; i++) {
        yield x[i];
    }
    yield 7;
}
let fun = gen(numbers);
console.log(fun.next().value);//1
console.log(fun.next().value);//2
console.log(fun.next().value);//3
console.log(fun.next().value);//4
console.log(fun.next().value);//5
console.log(fun.next().value);//6
console.log(fun.next().value);//7

//Ex2
function* test() {
    yield* [1, 2, 3, 4, 5, 6];
    yield 7
}
```

```
let ite = test();
console.log(ite.next().value);
console.log(ite.next().value);
console.log(ite.next().value);
console.log(ite.next().value);
console.log(ite.next().value);
console.log(ite.next().value);
console.log(ite.next().value);
```

* you can loop to the generator results of function
EX:

```
function* test() {
    yield "this is one";
    yield "this is two";
    yield "this is three";
    yield "this is four";
    yield "this is five";
    yield "this is six";
}

let ite = test();
for (let i of ite) {
    console.log(i);
}
```

* using return will stop the iteration of generator
EX:

```
function* test() {
    yield "this is one";
    yield "this is two";
    yield "this is three";
    return "this is four";
    yield "this is five";
    yield "this is six";
}

let ite = test();
for (let i of ite) {
    console.log(i);
}
```

*note : it’s not including number four , because return with generator will stop and not return any thing

* you can make infinite number generator
EX:

```
function* test() {
    let i = 0;
    while (true) {
        yield i++;
    }
}

let fun = test();
console.log(fun.next().value)
console.log(fun.next().value)
console.log(fun.next().value)
console.log(fun.next().value)
console.log(fun.next().value)
```

17-modules (import and export)
- Note : use live server extension

- you can link 2 pages or more together in js
- syntax:

-first link 2 pages in html:
EX:

```
<script src="test.js" type="module"></script>
<script src="test2.js" type="module"></script>
```

-Use export word next to code in 1st file
EX:

```
export const name1 = "ahmed";
export const x = () => {
    return 1;
}
```

-use import {} from ‘test2.js ‘

EX:

```
import { name1, x } from './test.js';
console.log(name1);
console.log(x());
```

- you can make export in one object
EX:

```
const name1 = "ahmed";
export const x = () => {
    return 1;
}
export { name1, x };
```

- you can change your export names before exporting
EX:

```
export {
    name1 as fName
    , x as Xfunc
};
```

```
import { fName, Xfunc } from './test.js';
console.log(fName);
console.log(Xfunc());
```

- you can import all by using * and will store in object
EX:

```
const name1 = "ahmed";
export const x = () => {
    return 1;
}
const arr = [10, 20, 30, 40]
export {
    name1 as fName,
    x as Xfunc,
    arr as numbers
};
```

```
import * as all from './test.js';
console.log(all.fName);
console.log(all.Xfunc());
for (let i = 0; i < all.numbers.length; i++) {
    console.log(all.numbers[i]);
}
```

* every page.js have one default export (if you tried make another one I will cause error)
EX:

```
export const name1 = "ahmed";
export default function test() {
    return "this is test";
}
```

* you can call it here with any name you want
--

```
import * as all from './test.js';
console.log(all.name1);
import x from './test.js'
console.log(x());
```