SZERVER OLDALI JAVASCRIPT

2. hét

Javascript nyelvi elemek

NYELVI ALAPOK:

Ez sajnos igen száraz anyag,

Viszont a megértékhez és a nyelv elsajátításához kell.

Próbá[illegible]eg random gifekkel feldobni.

MIRŐL LESZ SZÓ

- Alapok
- Típusok
- Osztályok
- Függvények
- Closure

ALAPOK: HOGY NÉZ KI A KÓD

```
var valtozo = 'ertek'; // ez egy comment
let masodik = "ertek2";
const harmadik = 3;

/*
 Tobbsoros comment
 */

 function egyfv (parameter1, paramter2){
     return 'visszateresi ertek';
 }
```

ALAPOK: OPERÁTOROK

Javascript gyengén típusos nyelv, lényegében vannak típusok, de minden konverzió automatikusan történik.

Az ellenőrzéseknél oda kell figyelni, hogy érték szerint egyenlő (**==**) vagy típus és érték szerint egyenlő (**===**) alapján ellenőrzünk.

Még értelmezett operátorok: != , !== , > , < , >= , <=

ALAPOK: IF

Az **if**, **while**, **switch** (típusos egyenlőségként) a megszokotthoz hasonlóan működik.

```
const a = '2';
const b = 2;

if (a == b) {}              //true
if (a === b) {}           //false
console.log('3' > 2)      //true
if (valami){
}else if (valami2){
}
```

ALAPOK: SWITCH

Switch típusosan ellenőriz.

```
const b='0';

switch (b){
  case 0:
    console.log('oh');
    break;
  case '0':
    console.log('hopp');
    break;
  default:
    console.log('meh');
    break;
}
```

ALAPOK: WHILE

```
let b=1;
while (b!==3){
    console.log(b);         //1,2,3
    b+=1;
}
```

TÍPUSOK

C++:

```
int szam = 2;
szam = 'valami';  // Error

string szoveg = new std::string('boo');

void valami = szoveg + szam; // Error
```

Javascript:

```
let szam = 2;
szam = 'valami';

const szoveg = 'boo';
let valami = szoveg + szam; //'boovalami' vagy
//ha nincs a 2. sor, akkor 'boo2'
```

TÍPUSOK

Egyszerű írni és olvasni!

- automatikus típus konvertálások
- nincs memória kezelés (foglalás / felszabadítás)
- nincs referencia kezelés (később kitérünk rá)

Viszont nem jelzi fordítási időben a hibákat, a működés nem mindig az, mint aminek kellene lennie

TÍPUSOK

Érdemes a típusokra figyelni.

```
const a = 2;
const b = '2';
console.log(a+b);   // 22  (string)
console.log(a/b);   // 1     (number)
console.log(a*b);   // 4    (number)
```

TÍPUSOK

Hogyan tudom meg, hogy mi milyen típus?

```
const a = 2;
const b = 'valami';
let c;

console.log(typeof a);    //number
console.log(typeof b);    //string
console.log(typeof c);    //undefined
```

TÍPUSOK

Alaptípusok:

- Undefined
- Null
- Boolean
- String
- Number
- Object

TÍPUSOK

Pár "nem létező" osztály, ami a nyelv megértéséhez elengedhetetlen:

- Reference
- List
- Completion (erről inkább nem beszélek, break, continue, return, throw részeknél izgalmas)

UNDEFINED

A nyelv dinamikus, tehát egy adott ponton egy változó akár lehet még nem definiált (akár globálisan is).

A globális névtérben van egy **undefined** változó definiálva, aminek a típusa **Undefined**.

```
const b = {};
console.log(typeof b.a);    //undefined
console.log(typeof undefined); //undefined
const a = 2;
console.log(typeof a);    //number (hiszen az előbb definiáltuk)
```

UNDEFINED

Az **undefined** globális változó (vagy bármi, ami **definiált** de a típusa undefined) részt vehet műveletekben.

```
console.log(undefined + 'alma')    ;    //'undefinedalma'
console.log(undefined + 3);            //NaN (ez egy number)

const myun = undefined;
if (myun === undefined) {}    // true

// ReferenceError: nemletezo is not defined
if (nemletezo === undefined) {}
```

UNDEFINED

Hogyan ellenőrzöm, hogy létezik az adott változó?

```
if (typeof myun !== 'undefined'){
// ...
}
```

NULL

C++ irányból jövőknek kell megnyutatásként, hogy van valami, ami nem 0 de mégis "valami". Undefinedhoz hasonló, és a szabvány miatt a *typeof* értéke object.

```
const e = null;
console.log(e == 0);         //false
console.log(e == false);     //false
console.log(e == true);         //false
```

Ritkán használjuk.

BOOLEAN

Két értéke lehet, **true** és **false**.

A kapcsolata a többi osztállyal kissé kaotikus:

```
const u = undefined;
console.log(false == u, true == u, !u, !!u)                    //false f
console.log(false == null, true == null, !null, !!null)        //false f
console.log(false == 0, true == 0, !0, !!0)                    //true fa
console.log(false == 11, true == 11, !11, !!11)                //false f
console.log(false == '', true == '', !'', !!'')                //true fa
console.log(false == 'a', true == 'a', !'a', !!'a')            //false f
console.log(false == [], true == [], ![], !![])                //true fa
console.log(false == ['a'], true == ['a'], !['a'], !!['a'])    //false f
console.log(false == {}, true == {}, !{}, !!{})                //false f
console.log(false == {a:1}, true == {a:1}, !{a:1}, !!{a:1})    //false f
```

WTF!

If a value can be converted to true, the value is so-called truthy. If a value can be converted to false, the value is so-called falsy.

All values are truthy unless they are defined as falsy (i.e., except for false, 0, -0, 0n, "", null, undefined, and NaN).

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_NOT

https://developer.mozilla.org/en-US/docs/Glossary/Falsy

https://developer.mozilla.org/en-US/docs/Glossary/Truthy

STRING

Szövegek tárolását oldja meg, nagyon sok kiegészítő funkcióval. Primitív típus, de osztályként viselkedik, amikor műveleteket végzünk rajta. A + concatenál, ami a stringgé való konvertálást is elintézi.

A ' és " közül bármelyikkel definiálható.

```
const s = 'Lorem ipsum';
const g = "ipsum";

console.log(s);                 // Lorem ipsum
console.log(s+g);              // Lorem ipsumipsum
console.log(s+42+g);          // Lorem ipsum42ipsum
```

STRING

- .indexOf(string) : az adott szövegben megkeresi a másik első előfordulását, ha létezik
- .substr(from,count): a szöveg egy részét adja vissza, a from karaterről count számban

```
const s = 'Lorem ipsum';
const g = "ipsum";

console.log(s.indexOf(g))         // 6
console.log(s.indexOf('boo'))     // -1

console.log(s.substr(1,3));         //ore
```

NUMBER

Számok tárolására való, gyakorlatilag a float-nak felel meg (de intet is tárolhat). A műveletek a Number "terén" belül maradnak általában, bármit is tesztünk.

```
const i = 2;
const j = 5.222;

console.log(i);             //2
console.log(j);             //5.222
console.log(i+j);          //7.222
console.log(i/3);          //0.6666666666666666
console.log(i/0);          //Infinity , de nem dob hibát!!!
console.log(i/'2');          //1
console.log(i/'2a');     //NaN (Not-a-Number)

//number, tehát typeof NaN === "number"...
console.log(typeof(i/'2a'));

console.log(typeof(i/0));     //number
console.log(typeof(i+j));     //number
```

NUMBER

Ezen az objektumon nem definiáltak osztály típusú műveleteket, e helyett a *Math* segédosztályt kell használni.

FONTOSABB RÉSZEI:

- Math.round(e) - kerekítés
- Math.floor(e) - csonkolás
- Math.random() - 0,1 közötti random szám

```
const e = 5.6;

console.log(Math.round(e));         //6
console.log(Math.floor(e));         //5
console.log(Math.random());         //0.800291629973799
```

NUMBER

Stringből hogyan lesz number biztonságosan?

- parseInt(string,radix)
- parseFloat(string)

```
parseInt("10")                 //10
parseInt("10.00")             //10
parseInt("10.33")             //10
parseInt("34 45 66")         //34
parseInt(" 60 ")             //60
parseInt("40 years")         //40
parseInt("He was 40")         //NaN

parseFloat("10")             //10
parseFloat("10.00")             //10
parseFloat("10.33")             //10.33
parseFloat("34 45 66")         //34
parseFloat(" 60 ")             //60
parseFloat("40 years")         //40
parseFloat("He was 40")         //NaN
```

LIST

Typeof szerint nincs külön lista típus, de a valóságban lényegében van. Nagyon sokszor használjuk, így egyszerű definiálni, illetve FIFO és FILO módban is nagyon jól tud működni.

LIST

- lista[i]: i. elem kinyerése, 0-tól indulva
- lista.push(e1,e2,e3,...): elem hozzáadása a lista végéhez
- lista.pop(): lista végén lévő elem lekérdezése és eltávolítása a listából
- lista.shift(): első elem levétele és eltávolítása

```
let lista = [];  //üres lista

lista.push(1);            //lista: [1]
lista.push(2,3);        //lista: [1,2,3]
console.log(lista[1]);    //2

let elem = lista.pop();    //elem:3 , lista: [1,2]
elem = lista.shift();    //elem:1 , lista: [2]
console.log(typeof lista); //object
```

LIST

- forEach(cb,thisObject): minden elemen sorban végrehajtunk egy műveletet
- indexOf(e): e elem első előfordulásának megkeresése (mint string)

```
let lista = [1,2,3,'4pista'];
let ret = [];

lista.forEach(function(e){
    ret.push(parseInt(e)*2);
});

console.log(ret);                  // [2, 4, 6, 8]

console.log(ret.indexOf(4));     //1
console.log(ret.indexOf('4'));     //-1
```

OBJECT

A legfontosabb, hogy az Object itt nem Class, definíció szerint:

Object is an unordered collection of key-value pairs.

Bizonyos nyelvekben ez a map, viszont javascriptben, majd látjuk, erősen keveredik az Object az OOP dolgokkal.

OBJECT

Az object egy dinamikus dolog, tehát bármikor bármelyik részét módosíthatjuk, vagy módosulhat, létrejöhet vagy törlődhet.

ELŐSZÖR DEFINIÁLJUK:

```
let emptyObj = {};    //tagváltozó nélküli objektum
let kutya = {
    lab: 4,
    fej: 1,
    ugat: function(){
        console.log('Nyauuu');
    }
};
console.log(kutya.lab);            //4
console.log(kutya.fej + 3);     //4
kutya.ugat();                       // Nyauuu
console.log(kutya['lab']);         //4: igen, így is elérhető ez
```

OBJECT

Mivel dinamikus, bármikor módosíthatok bármit.

```
kutya.fej=3;
console.log(kutya.fej);     //3

kutya.ugat = function(){ console.log('Wuf'); }
kutya.ugat();                //Wuf

//törölhetőek is a tagváltozók
delete kutya.lab;
console.log(kutya.lab);        //undefined

console.log(typeof(kutya));        //object
```

Ugye hogy mennyire "majdnem" OOP.

OBJECT

Az objectek magukra a *this* -el tudnak hivatkozni, ami megint csak OOP szerű.

```
const kutya = {
    say: 'Wuf',
    ugat: function(){
        console.log(this.say);
    }
};
kutya.ugat();    //Wuf
```

De ez még mindig nem osztály típusú működés, nincs konstruktor, nincs semmi. Mégis hogyan lesz ebből osztály?

CLASS

```
class Kutya {
  constructor(say) {
    this.say = say;
  }
  ugat() {
    console.log(this.say);
  }
}
const kutya = new Kutya("Wuf");
kutya.ugat();
```

ARROW FUNCTION

```
const nums = [1,2,3,4,5,6,7,8];
const one  = nums.map(v => v + 1)
const two = nums.map(v => { return v + 1;})
const three  = nums.map((v, i) => v + i)
console.log(nums);
console.log(one);
console.log(two);
console.log(three);
```

Lexical this!!!!

SPREAD OPERATOR

```
const a = [ "hello", true, 7 ]
const b = [ 1, 2, ...a ];
console.log(b);
const c = { a: "hello", b: true, c: 7 }
const d = {d: "foo", ...c }
console.log(d);
```

FÜGGVÉNYEK

Mivel az OOP szemlélet ennyire kaotikus, így nagyon nagy hangsúly helyeződik a függvényekre és főleg a callbackekre az aszinkron működés miatt.

EGYSZERŰ ÖSSZEADÁS

```
function osszead(a,b){
    return a + b;
}

osszead(1,3);    //4
```

FÜGGVÉNYEK

A változók definiálására figyelni kell, a változó értelmezésének terére (scope). **Ezt kerülni kell, mert a nem definiált változók a globális térben kerülnek létrehozásra!!**

EGYSZERŰ ÖSSZEADÁS

```
function osszead(a,b){
    c = a + b;
    return c;
}

osszead(1,3);          //4
console.log(c);          //4 !!!!
```

Ez BUG, nem FEATURE!!

FÜGGVÉNYEK

Maguk a függvények is elmenthetőek változókba,illetve az egyik függvény ezáltal átadható a másiknak.

```
const osszead = function (a,b){
    return a + b;
}

osszead(1,3);    //4
```

FÜGGVÉNYEK

```
function muvelet(a,b,amit){
    console.log("a: " + a + " b: "+ b);
    console.log("eredmeny: " + amit(a, b));
}
const osszead = function (a,b){
    return a + b;
}
const kivon = function (a,b){
    return a - b;
}
muvelet(1,3,osszead);    //a: 1 b: 3 eredmeny: 4
muvelet(1,3,kivon);        //a: 1 b: 3 eredmeny: -2
```

Függvény paraméterként átadva

FÜGGVÉNYEK

Függvények amik függvényeket adnak vissza. :)

```
function ugat(mit) {
    return function(){
        console.log(mit);
    };
}

const valami = 'vau';
const wuf = ugat(valami);
wuf();
```

FÜGGVÉNYEK

Viszont mivel nincs referencia / érték alapú átadás, csak paraméter átadás, mi történik, ha valamelyik paramétert megváltoztatom futás közben?

CLOSURES

Closures are functions that refer to independent (free) variables. In other words, the function defined in the closure 'remembers' the environment in which it was created.

Most tessék kapaszkodni.

CLOSURES BEVEZETÉS

setTimeout(fv, ms) itt a ms késleltetés millisecben.

Nézzünk egy egyszerű példát:

```
let a=0;

setTimeout(function () {
    a+=10;
    console.log('Hopp: ' + a);
},1000);

a+=1;
console.log('Kopp: ' + a);
```

CLOSURES BEVEZETÉS

Milyen sorrendben futnak le a parancsok?

```
let a=0;                        //1

setTimeout(function () {
    a+=10;                       //5
    console.log('Hopp: ' + a);    //6
},1000);                        //2, a setTimeout hívás
//, nem a belső függvény

a+=1;                           //3
console.log('Kopp: ' + a);     //4
```

CLOSURES BEVEZETÉS

És ha ezt a kódot nézzük?

```
function belso(a){
    return function () {
        a+=10;
        console.log('Hopp: ' + a);
    };
}
let a=0;
setTimeout(belso(a),1000);
a+=1;
console.log('Kopp: ' + a);
```

CLOSURES BEVEZETÉS

Egy extra sor beszúrásával árnyaltabb lesz a kép. Futási sorrend jelezve van megint.

```
function belso(a){
    let b=a;                        //3
    return function () {
        b+=10;                        //6
        console.log('Hopp: ' + b);    //7
    };
}
let a=0;                            //1
setTimeout(belso(a),1000); //2, belső fv lefut, egy fv-t ad vissza
a+=1;                               //4
console.log('Kopp: ' + a);          //5
```

CLOSURES BEVEZETÉS

```
function belso(a){
    return function () {
        a.ertek+=10;
        console.log('Hopp: ' + a.ertek);
    };
}
let a={ertek:0};
setTimeout(belso(a),1000);
a.ertek+=1;
a = {ertek:2};                          //ez a lényeg
console.log('Kopp: ' + a.ertek);
```

HOGY IS VAN EZ?

A belső függvények olyan külső (akár globális akár lokális) változókra hivatkoznak, amikkel a függvény létrehozása és meghívása során bármi történhet. Így a javascript engine a változókat "megpróbálja megtartani" referenciaként.

Objectek esetében ha az őket tartalmazó változót módosítom, akkor lényegében új objektumot hozok létre, és nem az eredeti referencia értékét módosítom.

Egyszerű típusok (string,number) esetében ezzel nincs gond, mert nem hozok létre új objektumot, hanem a meglévő értékét módosítom.

Ez látszik az előző dián.

LÉNYEGÉBEN ENNYI

Erre épül a nyelv, persze, sok extrával és hasznos kis aprósággal, de a többit majd csakorlatban.

Köszönöm a kitartást!