

# Web technológia

## JavaScript, 1. rész

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

[https://github.com/wajzy/GKxB\\_INTM049.git](https://github.com/wajzy/GKxB_INTM049.git)

2022. január 6.

## Jellemzők:

- Egyetlen típus létezik csak: 64 bites lebegőpontos ábrázolás
- Pl. 42, 12.34, -34.56, 1e3, -1e3, 1e-3, -1e-3, -1.23e-4, -1.23E+4, ...
- Különleges értékek: Infinity, -Infinity, NaN
- Pl.  $0/0 \rightarrow \text{NaN}$ ,  $1/0 \rightarrow \text{Infinity}$

## Operátorok (Precedencia táblázat)

+  $5+3 \rightarrow 8$

−  $5-3 \rightarrow 2$

×  $5*3 \rightarrow 15$

/  $5/3 \rightarrow 1.6666666666666667$

%  $5\%3 \rightarrow 2$ ,  $-5\%3 \rightarrow -2$ ,  $5\%-3 \rightarrow 2$

\*\*  $5**3 \rightarrow 125$

+ "Java" + 'Script' → "JavaScript"

## Jellemzők

- Értékek: `true`, `false`
- Pl. `5 < 3` → `false`

## Logikai operátorok

és `true && false` → `false`

vagy `true || false` → `true`

nem `!true` → `false`

*Short circuit evaluation* (pl. alapérték megadására):

`undefined || "Gizi" → "Gizi"`, `null || "Gizi" → "Gizi"`,  
`"" || "Gizi" → "Gizi"`, `"Gizi" || "Mari" → "Gizi"`

## Relációs operátorok

- `==, !=, <, <=, >, >=`
- Pl. `"Bill" != "Gates" → true`, `Infinity == Infinity → true`,  
`de NaN == NaN → false`
- Karakterláncok összehasonlítása: karakterkódok alapján

Üres értékek: valaminek a hiányát jelzik

- undefined
- null

Egyoperandusú operátorok

típus `typeof(5) → "number"`, `typeof("Gizi") → "string"`

— `-(5) → -5`

Háromoperandusú operátor

? `1<2?"kisebb":"nagyobb" → "kisebb"`

Néhány példa:

- `5 * null → 0`
- `"5" - 3 → 2`
- `"5" + 3 → "53"`
- `"öt" * 3 → NaN`, `5 * undefined → NaN`
- `false == 0 → true`, `true == 1 → true`, `true == 2 → false`,  
`"" == false → true`
- Definiált az érték? `null == undefined → true`, `null == 0 → false`

Típusok egyezését megkövetelő operátorok: `===`, `!==`

## Változók (*variable*, *binding*)

- Deklaráció: `let` (blokk hatáskör), `var` (függvény hatáskör)
- Inkább tekinthető értékre mutató referenciának, mint valódi tárolónak

### Példa

```
a → ReferenceError: a is not defined
let a
a → undefined
a = 5
a → 5
let b = 3, c
a * b → 15
```



## Konstansok

- `const`

### Példa

```
const c = 3.14
```

```
c = 2 → TypeError: invalid assignment to const 'c'
```

## Névadási szabályok

- betűket, számokat, \$ és \_ karaktereket tartalmazhat
- számjeggyel nem kezdődhet
- nem lehet foglalt szó (pl. let)
- kis- és nagybetűket megkülönbözteti
- javasolt stílus: *camel case* (hosszuValtozoNeve)

## Változókkal használható (összetett és unáris) operátorok

- +=, -=, \*=, /=, %=, &&=, ||=, \*\*=, ...
- ++, -

## Környezet (*environment*)

- adott pillanatban létező változók és értékeik
- gyakorlatilag soha nincs üres környezet

## Megjegyzések

- // egysoros
- /\* több  
soros \*/

## Szelekció

- `if(feltétel) utasítás;`
- `if(feltétel) {  
    // utasítások  
}`
- `if(feltétel) {  
    // igaz ág utasításai  
} else {  
    // hamis ág utasításai  
}`

- Mikor **nem** teljesül a *feltétel*?

- `false`
- `0`
- `""`
- `NaN`
- `null`
- `undefined`

## Több irányú elágazás

```
switch(kifejezés) {  
    case érték1:  
        // utasítások  
        break;  
    case érték2:  
    case érték3:  
        // utasítások  
        break;  
    default:  
        // utasítások  
        break;  
}
```

Az értéknek és a típusnak is egyeznie kell!  
A *default* ág elhagyható.

## Ciklusok

```
for(előkészítés; ismétlési_feltétel; frissítés) {  
    // Ciklusmag utasításai  
}
```

```
while(ismétlési_feltétel) {  
    // Ciklusmag utasításai  
}
```

```
do {  
    // Ciklusmag utasításai  
} while (ismétlési_feltétel);
```

break, continue

## Háromszög rajzolás (megoldás)

A böngésző JavaScript konzolján egy sornyi szöveget a `console.log()` hívással tud megjeleníteni. Használja ezt a következő háromszög megrajzolására:

```
*  
**  
***  
****  
*****
```

## X rajzolás (megoldás)

Most rajzoljon 5x5-ös méretű X-et csillagokból:

```
*      *  
  *    *  
    *  
  *    *  
*      *
```

## Sakktábla (megoldás)

Rajzoljon meg egy 8x8-as méretű sakktáblát, szintén csillagokból!

```
* * * *
* * * *
 * * * *
* * * *
 * * * *
* * * *
 * * * *
* * * *
```



## FizzBuzz (megoldás)

Vizsgálja meg az egész számokat 1-től 100-ig, majd a vizsgálat eredményét jelenítse meg egymás alatti sorokban! Ha a szám osztható 3-mal, írja ki, hogy *Fizz*, ha 5-tel osztható, akkor azt, hogy *Buzz*, ha pedig 3-mal és 5-tel is osztható, akkor azt, hogy *FizzBuzz*! Ha egyik számmal sem osztható, akkor írja ki a vizsgált számot!

```
1
2
Fizz
4
Buzz
Fizz
...
```

Definíció: a függvény, mint *érték* jelenik meg (**hatvanyDef.js**)

```
1  const hatvany = function(alap , kitevo) {  
2      let h = 1;  
3      for(let k=1; k<=kitevo; k++) {  
4          h *= alap;  
5      }  
6      return h;  
7  }  
8  
9  console.log(hatvany(2, 0)); // 1  
10 console.log(hatvany(2, 1)); // 2  
11 console.log(hatvany(2, 2)); // 4  
12 console.log(hatvany(3, 2)); // 9
```

```
console.log(hatvany(2, 0)); // 1
console.log(hatvany(2, 1)); // 2
console.log(hatvany(2, 2)); // 4
console.log(hatvany(3, 2)); // 9

function hatvany(alap, kitevo) {
    let h = 1;
    for(let k=1; k<=kitevo; k++) {
        h *= alap;
    }
    return h;
}
```

Nyíl (*arrow*) függvény: tömörebb megadás (*hatvanyNyil.js*)

```
1  const hatvany = (alap , kitevo) => {  
2    let h = 1;  
3    for(let k=1; k<=kitevo; k++) {  
4      h *= alap;  
5    }  
6    return h;  
7  }  
8  
9  console.log(hatvany(2, 0)); // 1  
10 console.log(hatvany(2, 1)); // 2  
11 console.log(hatvany(2, 2)); // 4  
12 console.log(hatvany(3, 2)); // 9
```

## Nyíl függvények

- Ha pontosan egy paramétert fogad, a paraméterlista körüli zárójelek elhagyhatóak
- Ha egyetlen paramétert sem fogad, üres zárójelpár jelzi a paraméterlistát
- Ha a függvény teste egyetlen kifejezés értékét szolgáltatja, a `return` és a blokk elhagyható

### nyilValtozatok.js

```
1  const negyzet = alap => alap*alap;  
2  console.log(negyzet(3)); // 9  
3  
4  const udvozol = () => console.log("Szia!");  
5  udvozol(); // Szia!
```

## hatokor.js

```
1  let a = 1; // globális
2  {
3    let a = 2; // elfedés, lokális
4    let b = 3; // lokális
5    var c = 4; // globális
6    console.log('a=${a}, b=${b}, c=${c}'); // a=2, b=3, c=4
7  }
8  //console.log('a=${a}, b=${b}, c=${c}'); // ReferenceError: b is not defined
9  console.log('a=${a}, c=${c}'); // a=1, c=4
```

## Paraméterezés

- Nem ellenőrzi híváskor sem a paraméterek számát, sem azok típusát! → felesleges paramétereket figyelmen kívül hagyja, a hiányzók értéke `undefined`
- A `return` nélküli, vagy a `return` után kifejezést nem tartalmazó függvények visszatérési értéke `undefined`
- Tetszőleges számú paramétert fogadó fv. is készíthető (ld. később)

### parameter1.js

```
1 const negyzet = alap => alap*alap;  
2 console.log(negyzet(3, 4, 5)); // 9  
3 console.log(negyzet(3)); // 9  
4 console.log(negyzet()); // NaN  
5 console.log(negyzet("Micimackó")); // Nan
```

## parameter2.js

```
1  const negyzet = alap => {  
2    if (typeof(alap)=="number") {  
3      return alap*alap;  
4    } else {  
5      return;  
6    }  
7  }  
8  
9  console.log(negyzet(3)); // 9  
10 console.log(negyzet()); // undefined  
11 console.log(negyzet("Micimackó")); // undefined
```



A függvények *értékek*:

- függvények átadhatók más függvénynek paraméterként,
- függvény visszatérési értéke lehet függvény,
- függvény beágyazható másik függvénybe.

### paramFv1.js

```
1  const osszead = (a, b) => a + b;  
2  const muvelet = function(a, op, b) {  
3    console.log( `${a} + ${b} = ${op(a, b)} ` );  
4  }  
5  muvelet(3, osszead, 5); // 3 + 5 = 8
```

Névtelen (*anonymous*) függvények (paramFv2.js)

```
1  const muvelet = function(a, op, b) {  
2    console.log( `${a} + ${b} = ${op(a, b)} ` );  
3  }  
4  muvelet(3, (a, b) => a + b, 5); // 3 + 5 = 8  
5  muvelet(4,  
6    function(a, b) {  
7      return a + b;  
8    },  
9    7  
10 ); // 4 + 7 = 11
```

## Függvények definiálása és azonnali hívása (paramFv3.js)

```
1 (function(a, op, b) {  
2   console.log(`${a} + ${b} = ${op(a, b)} `);  
3 })(3, (a, b) => a + b, 5); // 3 + 5 = 8  
4  
5 ((a, op, b) => {  
6   console.log(`${a} + ${b} = ${op(a, b)} `);  
7 })(4, (a, b) => a + b, 7); // 4 + 7 = 11
```

## Zárványok (*closure*)

- Mi történik, ha egy *külső* függvény *lokális* változóit eléri egy *belső* függvény, amit *meghívunk* azután, hogy az őt létrehozó *külső* függvényből kiléptünk?

## Zárványok (*closure*)

- Mi történik, ha egy *külső* függvény *lokális* változóit eléri egy *belső* függvény, amit *meghívunk* azután, hogy az őt létrehozó *külső* függvényből *kiléptünk*?
- A függvény megőrzi futtatási környezetét

## Függvény, mint visszatérési érték (zarvany.js)

```
1  const hatvany = (kitevo) => {  
2    return alap => {  
3      let h = 1;  
4      for(let k=1; k<=kitevo; k++) {  
5        h *= alap;  
6      }  
7      return h;  
8    };  
9  };  
10 const negyzet = hatvany(2);  
11 const kob = hatvany(3);  
12 console.log(negyzet(3)); // 9  
13 console.log(kob(5)); // 125
```

## Rekurzív hatványozás (rekurzio.js)

```
1  const hatvany = function(alap, kitevo) {  
2      if(kitevo == 0) return 1;  
3      if(kitevo == 1) return alap;  
4      let h = hatvany(alap, (kitevo-kitevo%2)/2);  
5      h *= h;  
6      if(kitevo%2) {  
7          h *= alap;  
8      }  
9      return h;  
10 }  
11 console.log(hatvany(5, 3)); // 125
```

## Fibonacci-számok (fibonacci.js)

Fibonacci-sorozat: másodrendben rekurzív sorozat. Képzeltbeli nyúlcsalád növekedése: hány pár nyúl lesz  $n$  hónap múlva, ha

- az első hónapban csak egyetlen újszülött nyúl-pár van,
- az újszülött nyúl-párok két hónap alatt válnak termékennyé,
- minden termékeny nyúl-pár minden hónapban egy újabb párt szül,
- és a nyulak örökké élnek.

$$F_n = \begin{cases} 0, & \text{ha } n = 0 \\ 1, & \text{ha } n = 1 \\ F_{n-1} + F_{n-2} & \text{ha } n > 1 \end{cases}$$

Készítse el azt a fibonacci függvényt, melynek paramétere a sorozat valamely elemének indexe ( $n$ ), visszatérési értéke a sorozat megfelelő eleme!





## Szinusz függvény (sin.js)

Írja meg azt a `sin` függvényt, amely visszaadja a paraméterként kapott, radiánban mért szög szinuszát!

A keresett érték meghatározható a szinusz függvény sorba fejtésével:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \text{ azaz } \sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

A függvénynek természetesen nem kell végtelen sok tagot, illetve azok összegét meghatároznia. Elegendő, ha a függvény  $\epsilon = 10^{-6}$  pontossággal kiszámítja az eredményt.

## Objektumok

- tulajdonság (kulcs) – érték párok (csak a null-nak és az undefined-nak nincsenek tulajdonságai a nyelvben)
- minden tulajdonság egyedi az objektumban
- a tulajdonság lehet adat vagy függvény (metódus)
- a tulajdonságot az értéktől : választja el, a párokat egymástól ,

### Objektum definiálása literálként

```
1 const hg = {  
2   nev: "Kovács István",  
3   neptun: "a1b2c3",  
4   zh: 12  
5 };
```

hg kötése konstans, de ettől még a tulajdonságok értéke megváltoztatható.  
Tulajdonságok elérése: `objektum.tulajdonság` formában

### Objektum módosítása

```
6 console.log(hg.zh); // 12
7 hg.zh = 14; // Tulajdonságok változtathatók
8 console.log(hg.zh); // 14
9 // De const miatt az objektum nem váltható le
10 /*
11 hg = { // TypeError: invalid assignment to const 'hg'
12     nev: "Nagy Péter",
13     neptun: "1q2w3e",
14     zh: 13
15 };
16 */
```

## Két kötés (referencia) ugyanarra az objektumra

```
18 // Kötések (binding), nem klasszikus változók
19 let hallgato = hg;
20 hg.zh = 15;
21 console.log(hallgato.zh); // 15
```

### Tulajdonságok feltérképezése

- in (tartalmazás) operátor (vs. if(objektum.tulajdonság) ...)
- for/in ciklus, a tulajdonságokon történő iterálásra

Ha a tulajdonság neve kötéssel adott, a `.` operátor nem használható →  
objektum["tulajdonság"]

## Tulajdonságok elérése

```
23 // Tulajdonság létezésének tesztelése
24 console.log("nev" in hg); // true
25 console.log("evfolyam" in hg); // false
26
27 // Milyen tulajdonságok vannak az objektumban, milyen értékkel?
28 function nyomtat(obj) {
29     for(let tul in obj) {
30         console.log(tul, ":", obj[tul]);
31     }
32 }
```

## Objektumok tartalmának másolása:

```
Object.assign(cél, forrás1, forrás2, ..., forrásN)
```

Visszatérési érték: cél

### Tulajdonságok másolása

```
34 // Objektumok tartalmi másolása
35 const hg2 = {
36   nev: "Kovács Emőke",
37   zh2: 19
38 };
39
40 let egyesített = Object.assign({}, hg, hg2);
41 nyomtat(egyesített);
42 Object.assign(hg2, hg);
43 nyomtat(hg2);
```

### Kimenet

```
nev : Kovács Emőke
neptun : a1b2c3
zh : 15
zh2 : 19
```

```
nev : Kovács István
zh2 : 19
neptun : a1b2c3
zh : 15
```

Tulajdonságok értékadással bármikor felvehetők az objektumba, és delete operátorral törölhetők

### Tulajdonságok hozzáadása, törlése

```
45 // Tulajdonságok utólagos hozzáadása, elvétele
46 hg.zh1 = hg.zh;
47 delete hg.zh;
48 hg.zh2 = 20;
49 nyomtat(hg);
```

### Kimenet

```
nev : Kovács István
neptun : a1b2c3
zh1 : 15
zh2 : 20
```



Rövidített objektum definíciós szintakszis: a kötés neve lesz a tulajdonság neve is

### Metódus hozzáadása

```
51 let nev = "Fekete_Péter";  
52 let neptun = "abcdef";  
53 let zh1 = 12;  
54 let zh2 = 8;  
55 const hg3 = {  
56   nev: nev,  
57   neptun: neptun,  
58   zh1: zh1,  
59   zh2: zh2  
60 };  
61 nyomtat(hg3);  
62 const hg4 = { nev, neptun, zh1, zh2 };  
63 nyomtat(hg4);
```

### Kimenet

```
nev : Fekete Péter  
neptun : abcdef  
zh1 : 12  
zh2 : 8
```

```
nev : Fekete Péter  
neptun : abcdef  
zh1 : 12  
zh2 : 8
```

Metódus: a tulajdonság értéke függvény. Az objektum többi tulajdonsága a `this`-en keresztül érhető el

### Metódus hozzáadása

```
65 // Metódusok; arrow fn. nem használható,  
66 // mert nincs saját kötése a this-hez  
67 hg.getAlairas = function() {  
68     return (this.zh1+this.zh2) >= 20;  
69 }  
70 console.log(hg.getAlairas()); // true
```

## Tömbök

- Speciális objektumok, amelyekben a tulajdonságok nevei (kulcsok) nem negatív egész számok, de az értékek vegyesen bármilyen típusúak lehetnek
- Tömb literál létrehozása: [elem1, elem2, ..., elemN]
- Tömb elemszáma: tömb.length tulajdonság
- Elemek elérése: [] operátorral

### Tömb létrehozása, indexelés, elemszám megállapítás

```
1 let t1 = []; // üres tömb
2 console.log(typeof t1); // object
3 let t2 = ["Alma", "Banán", "Citrom"];
4 console.log(t2[1]); // Banán
5 t2[1] = "Burgonya";
6 console.log(t2[1]); // Burgonya
7 console.log(t2.length); // 3
```

A tömb bejárására használhatóak a `for/in` (tulajdonságok/indexek) és `for/of` (értékek) ciklusok

### Tömbök bejárása

```
9  function nyomtat1(tomb) {  
10    for(let elem of tomb) {  
11      console.log(elem);  
12    }  
13  }  
14  nyomtat1(t2);  
15  
16  function nyomtat2(tomb) {  
17    for(let idx in tomb) {  
18      console.log(idx, ":", tomb[idx]);  
19    }  
20  }  
21  nyomtat2(t2);
```

### Kimenet

```
Alma  
Burgonya  
Citrom  
  
0 : Alma  
1 : Burgonya  
2 : Citrom
```

Tömböt állít elő az `Object.keys()` egy objektum tulajdonságaiból

### Objektum tulajdonságainak visszaadása tömbként

```
23 let tulajdonsagok = Object.keys({  
24   egy: 1,  
25   ketto: 2,  
26   három: 3  
27 });  
28 nyomtat2(tulajdonsagok);
```

### Kimenet

```
0 : egy  
1 : ketto  
2 : három
```

További elemek hozzáadása egy kiválasztott indexű elemhez történő hozzárendeléssel lehetséges. A tömb elemszáma a legnagyobb index alapján kerül meghatározásra, **nem a tárolt elemek száma** alapján!

### Tömbök elemei

```
30 t2[3] = "Dió";  
31 nyomtat2(t2);  
32 t2[5] = "Füge";  
33 console.log(t2.length); // 6  
34 console.log(t2[4]); // undefined  
35 nyomtat2(t2);
```

### Kimenet

```
0 : Alma  
1 : Burgonya  
2 : Citrom  
3 : Dió  
6  
undefined  
0 : Alma  
1 : Burgonya  
2 : Citrom  
3 : Dió  
5 : Füge
```

A literál megadásakor is jelezhetjük, hogy bizonyos indexű elemeket nem kívánunk létrehozni.

### Hiányos tömbök

```
36 let t3 = ["Alma", , "Citrom", ];  
37 console.log(t3.length); // 3  
38 nyomtat2(t3);  
39 let t4 = ["Alma", , "Citrom", undefined];  
40 console.log(t4.length); // 4  
41 nyomtat2(t4);
```

### Kimenet

```
3  
0 : Alma  
2 : Citrom  
4  
0 : Alma  
2 : Citrom  
3 : undefined
```

## Tömbelem törlése: delete operátorral

### Tömbelem törlése

```
43 delete t4[2];  
44 nyomtat2(t4);
```

### Kimenet

```
0 : Alma  
3 : undefined
```



## Verem műveletek

- Tömb végén: `push()`/`pop()`
- Tömb elején: `unshift()`/`shift()` (vagyis egy *igazi* sort pl. a `push()`/`shift()` párossal lehetne létrehozni)

### Veremműveletek

```
46 let t5 = [ 1, 2, 3 ];
47 t5.push(4);
48 nyomtat2(t5);
49 console.log(t5.pop()); // 4
50 nyomtat2(t5);
51 t5.unshift(0);
52 nyomtat2(t5);
53 console.log(t5.shift()); // 0
54 nyomtat2(t5);
```

### Kimenet 1/2

```
0 : 1
1 : 2
2 : 3
3 : 4
4
0 : 1
1 : 2
2 : 3
```

### Kimenet 2/2

```
0 : 0
1 : 1
2 : 2
3 : 3
0
0 : 1
1 : 2
2 : 3
```

## Tömbök egyesítése: concat()

## Tömbök egyesítése

```
56 let t6 = ["Alma", "Banán"];  
57 let t7 = [1, 2, 3];  
58 let t8 = t6.concat(t7);  
59 nyomtat2(t8);
```

## Kimenet

```
0 : Alma  
1 : Banán  
2 : 1  
3 : 2  
4 : 3
```

Tömbelemek kivágása és beillesztése az eredeti tömb módosításával (= helyben):

- `tömb.splice(tol[, db[, elem1[, elem2[, ...[, elemN]]]])`
- `tol`: a műveletvégzés indexe, lehet negatív is
- `db`: a törölni kívánt elemek száma
- `elem1, elem2, ..., elemN`: beszúrandó új elemek
- visszatérési érték: a törölt elemek tömbje

### Törlés és beszúrás

```
61 let t9 = ["Alma", "Banán", "Citrom", "Dió"];
62 console.log(t9.splice(1, 2)); // ["Banán", "Citrom"]
63 console.log(t9); // ["Alma", "Dió"]
64 console.log(t9.splice(2, 0, "Eper", "Füge")); // []
65 console.log(t9); // ["Alma", "Dió", "Eper", "Füge"]
66 console.log(t9.splice(-1, 1)); // ["Füge"]
```

## Új tömb létrehozása meglévő tömb elemeinek kimásolásával

- `tömb.slice(tol[, ig])`
- `tol`: kezdőindex
- `ig`: végindex (ezt már **nem** érinti a művelet); alapértelmezett értéke `tömb.length`
- az indexek lehetnek negatívak is
- visszatérési érték: az új tömb

### Új tömb létrehozása meglévő alapján

```
68 let t10 = ["Alma", "Banán", "Citrom", "Dió"];
69 //           0         1         2         3
70 //          -4        -3        -2        -1
71 console.log(t10.slice(1, 2)); // ["Banán"]
72 console.log(t10); // ["Alma", "Banán", "Citrom", "Dió"]
73 console.log(t10.slice(-3, -1)); // ["Banán", "Citrom"]
```

## Keresés tömbökben:

- `tömb.indexOf(keresett[, tol])`  
`tömb.lastIndexOf(keresett[, tol])`
- `indexOf`: balról jobbra, `lastIndexOf`: jobbról balra keres
- `keresett`: a keresett érték
- `tol`: keresés megkezdésének helye, index; elhagyható, és lehet negatív is
- visszatérési érték: -1, ha nincs találat

## Keresés tömbökben

```
75 let t11 = ["Alma", "Banán", "Citrom", "Alma"];
76 //      0      1      2      3
77 //     -4     -3     -2     -1
78 console.log(t11.indexOf("Banán")); // 1
79 console.log(t11.indexOf("Dió")); // -1
80 console.log(t11.lastIndexOf("Alma")); // 3
81 console.log(t11.indexOf("Alma", 1)); // 3
82 console.log(t11.indexOf("Alma", -3)); // 3
```

## Többdimenziós tömb egydimenziós tömbök egymásba ágyazásával hozható létre

### Többdimenziós tömbök

```
84 let t12 = [ ["Alma", "Banán"],  
85             [1, 2, 3] ];  
86 function mtxNyomtat(mtx) {  
87     for(let sor in mtx) {  
88         for(let cella in mtx[sor]) {  
89             console.log(sor, cella, ":", mtx[sor][cella]);  
90         }  
91     }  
92 }  
93 mtxNyomtat(t12);
```

### Kimenet

```
0 0 : Alma  
0 1 : Banán  
1 0 : 1  
1 1 : 2  
1 2 : 3
```

## Dekompozíció (destructuring)

Egy tömb elemeinek elérése nehézkes lehet, főleg ha többdimenziós, nagy elemszámú tömbről van szó. Példa: 2x2-es mátrix **determinánsának** meghatározása.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = a \times d - b \times c$$

Egyszerűbb, kifejezőbb, ha indexelés nélkül, közvetlenül elérhetők a mátrix elemei.

## Dekompozíció

```
1 function det1(mtx) {  
2   return mtx[0][0]*mtx[1][1] - mtx[0][1]*mtx[1][0];  
3 }  
4 const m = [[3, 5], [2, 7]];  
5 console.log(det1(m)); // 11  
6 const det2 = ([[a, b], [c, d]]) => a*d - b*c;  
7 console.log(det2(m)); // 11
```

Hasonló dekompozíció objektumokkal is megvalósítható.

### Dekompozíció

```
9  let {nev, zh1, zh2} = { nev: "Fehér Ilona", neptun: "QWERTZ",  
10                          zh1: 12, zh2: 19 };  
11  console.log(`${nev} összesen ${zh1+zh2} pontot ért el a ZH-kon.`)  
12  // Fehér Ilona összesen 31 pontot ért el a ZH-kon.
```



Egy függvény fogadhat előre meg nem határozott számú paramétert, melyeket az arguments tömb-szerű változón keresztül érhet el.

### Változó számú paraméter

```
1 function atlag1() {  
2   let osszeg = 0;  
3   for(let adat of arguments) {  
4     osszeg += adat;  
5   }  
6   return osszeg / arguments.length;  
7 }  
8 console.log(atlag1(1, 2, 3, 4)); // 2.5
```

Fejlettebb megoldás a maradék paraméterek (rest parameters) használata, mely a külön átadott aktuális paramétereket egy adott nevű tömbbe gyűjti.

### Változó számú paraméter

```
10 function atlag2 (...adatok) {  
11     let osszeg = 0;  
12     for(let adat of adatok) {  
13         osszeg += adat;  
14     }  
15     return osszeg / adatok.length;  
16 }  
17 console.log(atlag2(1, 2, 3, 4)); // 2.5
```

Ez a megoldás független értékeknek tömbbe foglalására, és tömb elemeinek különálló változókba helyezésére is lehetőséget ad.

### Független változók ↔ tömb

```
19 let szamok1 = [1, 2, 3, 4];  
20 console.log(atlag2(...szamok1)); // 2.5  
21 let szamok2 = [0, ...szamok1, 5, 6];  
22 console.log(szamok2); // [ 0, 1, 2, 3, 4, 5, 6 ]
```