

Web technológia

JavaScript, 1. rész

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM049.git

2022. november 15.

A JavaScript fontosabb tulajdonságai

- Web oldalba ágyazható, de **szerver (backend) oldalon** is használható
- Objektum-orientált, de **nincsenek osztályok** (objektum alapú, vagy prototípus alapú)
- Gyengén típusos, dinamikusan típusos (változók futásidőben típust válthatnak) → kényelmes, de veszélyes
- Böngésző által értelmezett (interpreter/JIT) nyelv (script)
- HTML DOM (Document Object Model) elérése
- Eseménykezelés
- Dokumentáció: **ECMA**, **Mozilla**

+ "Java" + 'Script' → "JavaScript"

Jellemzők

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

Logikai operátorok

`és` `true && false` \rightarrow `false`
`vagy` `true || false` \rightarrow `true`
`nem` `!true` \rightarrow `false`


```
?: 1<2?"kisebb":"nagyobb" → "kisebb"
```

Automatikus típuskonverzió (*Type coercion*)

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: `===`, `!==`

Változókkal használható (összetett és unáris) operátorok

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

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`

Short circuit evaluation (pl. alapérték megadására):

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

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.

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!

[illegible]

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
```

Deklaráció: helye a hatókörön belül bárhol lehet ([hatvanyDek.js](#))

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

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
```

Régi módszer hiányzó paraméterek kezelésére (**hatvanyAlapertelmezettRegi.js**)

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

Alapértelmezett paraméter érték (hatvanyAlapertelmezett.js)

```
1  const hatvany = function(alap, kitevo=1) {
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
12 console.log(hatvany(2, 2)); // 4
```

Tulajdonságok:

- Paraméterek átadása balról jobbra, akár az alapértelmezett értékek felülírásával is
- Alapértelmezett érték kiszámítható kifejezéssel, akár fv. hívással is
- Ezek minden egyes híváskor kiértékelődnek
- Minden, a paramétertől balra lévő további paraméter használható inicializálásra
- [További részletek](#)

A függvények értékek:

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

Magasabbrendű függvények (Higher-Order Functions, HOF): olyan függvények, melyek fv. paramétert fogadnak, vagy fv.-t adnak vissza

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
```

```
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
```


Currying: egy több paramétert fogadó függvény megvalósítása kevesebb paramétert fogadó magasabb rendű függvényekkel

```
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
```


Négyzetgyökvonás (gyok.js)

Készítse el a gyok függvényt, mely Newton módszerrel meghatározza és visszatérési értéként szolgáltatja paraméterének négyzetgyökét!

A módszer iteratív: egy sorozat egymást követő tagjait kell kiszámolni, melyek általában nagyon gyorsan konvergálnak a keresett eredményhez. A sorozat első elemét célszerű lenne a megoldás közeléből választani, de az egyszerűség kedvéért legyen ez nálunk mindig 10. Ha az utolsóként meghatározott tag értéke 10^{-6} -nál nem nagyobb mértékben tér el az utolsó előttiként kiszámolttól, akkor ezt az utolsóként kiszámolt értéket tekintjük a megoldásnak. A Newton módszer szerint a sorozat tagjait általánosan a következőképpen határozzuk meg:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Konkrétan a négyzetgyökvonás esetén, ha pl. az $x^2 = 612$ (itt 612 a gyok függvény aktuális paraméterének feleltethető meg) zérushelyét keressük, azaz $f(x) = x^2 - 612$ akkor $f'(x) = 2x$.

Ebből adódik, hogy $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 10 - \frac{10^2 - 612}{2 \cdot 10} = 35.6$ majd

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 35.6 - \frac{35.6^2 - 612}{2 \cdot 35.6} = 26.3955056, \text{ stb.}$$

(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:

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

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

Legnagyobb közös osztó (lnko.js)

Valósítsa meg az **Euklideszi algoritmust** két egész szám legnagyobb közös osztójának meghatározásához!

- 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 ,

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

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡


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

- `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 }
```

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

Visszatérési érték: cél

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
```

```
const hg5 = {
  nev: "Mezei_Virág",
  neptun: "1qay2w",
  zh1: 14,
  zh2: 19,
  getAlairas: function() { // metódus literálal
    return (this.zh1+this.zh2) >= 20;
  }
};
console.log(hg5.getAlairas()); // true
```


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
```

Objektum tulajdonságainak visszaadása tömbként

```
0 : egy
1 : ketto
2 : harom
```

Web technológiák - JavaScript

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
```


Tömbök egyesítése

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"]
```

A tömbök metódusai

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
    
```

Rendezés:

- `sort()`: karakterláncként kezelt adatokat rendez
- `sort(használatiFv)`: adott fv. által adott relációt figyelembe véve rendez

Rendezés különféle sorrendekbe

```

95 let tomb = [10, 2, 100, 20, 1, 200]
96 console.log(tomb.sort()) // 1, 10, 100, 2, 20, 200
97 console.log(tomb.sort((a, b) => {
98     if(a < b) {
99         return 1;
100     } else if(a == b) {
101         return 0;
102     } else {
103         return -1;
104     }
105 }))) // 200, 100, 20, 10, 2, 1

```

- `forEach()` minden egyes elemmel külön meghívja a paraméter fv.-t
- `every()` Igazat ad, ha a tesztelő fv. minden egyes elemre igazat ad → logikai és
- `some()` Igazat ad, ha a tesztelő fv. legalább egy elemre igazat ad → logikai vagy
- `filter()` Új tömböt készít és ad vissza, amely azokból az elemekből áll, melyekre a paraméter fv. igazat adott
- `map()` Minden elemet egyesével leképez egy újra, melyekből új tömböt állít elő
- `reduce()` Balról jobbra összevonja az elemeket egyetlen változóba
- `reduceRight()` Mint `reduce()`, csak jobbról balra haladva

Tömb fv. paramétert váró függvényei

```
107 let szamok = [10, 20, 30];
108 szamok.forEach(elem => console.log(elem / 10)); // 1 2 3
109 console.log(szamok.every(elem => elem > 15)); // false
110 console.log(szamok.some(elem => elem > 15)); // true
111 console.log(szamok.filter(elem => elem > 15)); // [20, 30]
112 console.log(szamok.map(elem => elem*elem)); // [100, 400, 900]
113 console.log(szamok.reduce((osszeg, szam) => osszeg + szam)); // 60
```

Többdimenziós tömb egydimenziós tömbök egymásba ágyazásával hozható létre

Többdimenziós tömbök

Kimenet

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

```
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);
```


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
```

Független változók \leftrightarrow 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 ]
```


atadas.js

```
1 function valtoztat(a, b, c)
2 {
3     a = a * 10;
4     b.tag = "megváltozott";
5     c = {tag: "megváltozott"};
6 }
7
8 var szam = 10;
9 var obj1 = {tag: "eredeti"};
10 var obj2 = {tag: "eredeti"};
11
12 valtoztat(szam, obj1, obj2);
13
14 console.log(szam); // 10
15 console.log(obj1.tag); // megváltozott
16 console.log(obj2.tag); // eredeti
```

Újradeklaráció: let/const esetén hiba, var/function esetén nem!

```

1  let a = 1;
2  // let a = 2; SyntaxError: redeclaration of let a
3  const b = 1;
4  // const b = 2; SyntaxError: redeclaration of const b
5  var c = 1;
6  var c = 2;
7  console.log(c); // 2
8
9  const muvelet = (n, m) => n + m;
10 //const muvelet = (n, m) => n * m; SyntaxError: redeclaration of const muvelet
11 function fv(n) {
12     return n + 1;
13 }
14 function fv(n) {
15     return n * n;
16 }
17 console.log(fv(3)); // 9
    
```

Kompozíció: függvények összefűzése, az egyik visszatérési értéke objektum, aminek szintén hívható egy függvénye

```

1  for(const idx of [1, 2, 3].keys()) { // iterátor a kulcsokra
2      console.log(idx);
3  } // 0 1 2
4
5  console.log(Array(5)); // 5 elemű üres tömböt hoz létre
6
7  const tartomany = n => [... Array(n).keys()];
8  const novel = a => a + 1;
9  const szorzas = (a, b) => a * b;
10 const faktorialis = (n) => tartomany(n).map(novel).reduce(szorzas);
11 console.log(faktorialis(5)) // 120
    
```

Typeof működése különböző típusú adatokkal → pl. fv. létezésének ellenőrzésére

```
1 console.log(typeof(console.log)); // function
2 console.log(typeof({kulcs: "ertek"})); // object
3 console.log(typeof([1, 2, 3])); // object
4 console.log(typeof(null)); // object
5 console.log(typeof(undefined)); // undefined
6 console.log(typeof(true)); // boolean
7 console.log(typeof(1)); // number
8 console.log(typeof("szó")); // string
```


Tömb kilapítása ([kilapitas.js](#))

Írjon olyan függvényt, ami egy mátrix összes elemét belerakja egy újonnan létrehozott egydimenziós tömbbe (vektorba)! Ötlet: `reduce()`, `concat()` metódusok használata.

Frobenius-norma ([frobenius.js](#))

Írjon olyan függvényt, ami kiszámolja egy mátrix [Frobenius-normáját](#)!

Mátrixok összevonása (hvstack.js)

Definiálja a `hstack()` függvényt, ami az azonos számú sorokból álló A és B mátrixok összevonásával elkészíti a C mátrixot a következőképpen:

$$hstack(A_{i,j}, B_{i,k}) = C \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,j} & B_{1,1} & B_{1,2} & \cdots & B_{1,k} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,j} & B_{2,1} & B_{2,2} & \cdots & B_{2,k} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ A_{i,1} & A_{i,2} & \cdots & A_{i,j} & B_{i,1} & B_{i,2} & \cdots & B_{i,k} \end{bmatrix}$$

Hasonlóan definiálja `vstack()`-et is, ami azonos számú oszlopot tartalmazó mátrixokkal végez műveletet:

$$vstack(A_{i,j}, B_{k,j}) = C \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,j} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ A_{i,1} & A_{i,2} & \cdots & A_{i,j} \\ B_{1,1} & B_{1,2} & \cdots & B_{1,j} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ B_{k,1} & B_{k,2} & \cdots & B_{k,j} \end{bmatrix}$$