Web technológia JavaScript, 1. rész

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM049.git 2023. november 2.







A JavaScript fontosabb tulaidonságai

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



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. $0/0 \rightarrow NaN$. $1/0 \rightarrow Infinity$

Operátorok (Precedencia táblázat)

$$+$$
 5+3 \rightarrow 8

$$-$$
 5-3 \rightarrow 2

$$\times$$
 5*3 \rightarrow 15

$$/$$
 5/3 \rightarrow 1.666666666666667



Jellemzők

- Unicode, 16 bit karakterenként
- Nincs specifikus típus egyetlen karakter tárolására
- Jelölés: '-ok vagy "-ek között
- Pl. 'JavaScript', "JavaScript", "Guns 'n' Roses", "Egy\nKettő\nHárom", 'Guns \'n\' Roses', "Új sor \\n megadásával kérhető."
- Template literal: '-ek között, kifejezések kiértékelése
- Pl. '5 * 3 = $\${5*3}$ ' \rightarrow "5 * 3 = 15"

Operátor



Jellemzők

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

Logikai operátorok

és true && false \rightarrow false

vagy true | | false → true

 $nem !true \rightarrow false$

Relációs operátorok

- **=** ==. !=. <. <=. >. >=
- Pl. "Bill" != "Gates" → true, Infinity == Infinity → true, de NaN == NaN → false (ld. isNaN(), isFinite())
- Karakterláncok összehasonlítása: karakterkódok alapján

Üres értékek: valaminek a hiányát jelzik

- undefined
- null

000000

Egyoperandusú operátorok

típus typeof(5)
$$\rightarrow$$
 "number", typeof("Gizi") \rightarrow "string" - -(5) \rightarrow -5

Háromoperandusú operátor

?: 1<2?"kisebb":"nagyobb"
$$\rightarrow$$
 "kisebb"

Néhány példa:

- \blacksquare 5 * null \rightarrow 0
- "5" 3 \rightarrow 2
- **■** "5" + 3 → "53"
- \blacksquare "öt" * 3 \rightarrow NaN, 5 * undefined \rightarrow NaN
- false == $0 \rightarrow \text{true}$, true == $1 \rightarrow \text{true}$, true == $2 \rightarrow \text{false}$. "" == false \rightarrow true
- Definialt az érték? null == undefined \rightarrow true, null == 0 \rightarrow 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

 $a \rightarrow undefined$

a = 5

 $a \rightarrow 5$

let b = 3. c

 $a * b \rightarrow 15$



Konstansok

const

Példa

const c =
$$3.14$$
 c = $2 \rightarrow \text{TypeError}$: 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

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

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

- 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.

```
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!

_

2

Fizz

4

Buzz

Fizz

. . .

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

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

Deklaráció: helye a hatókörön belül bárhol lehet (hatvany Dek. js)

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

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

```
const hatvany = (alap, kitevo) => {
     let h = 1:
     for(let k=1; k \le kitevo; k++) {
       h *= alap:
 5
 6
     return h:
 7
8
9
   console.log(hatvany(2, 0)); // 1
   console.log(hatvany(2, 1)); // 2
10
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
```

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

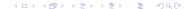
```
let a = 1; // globális
2
     let a = 2: // elfedés. lokális
     let b = 3: // lokális
     var c = 4: // globális
     console \log ('a=\$\{a\}, b=\$\{b\}, c=\$\{c\}'); // a=2, b=3, c=4
7
  //console.log('a=\$\{a\}, b=\$\{b\}, c=\$\{c\}'); // ReferenceError: b is not defined
8
   console.log('a=\{a\},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

Paraméterezés

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

```
parameter1.js
```

```
const negyzet = alap => alap*alap;
  console \log(\text{negyzet}(3, 4, 5)); // 9
  console.log(negyzet(3)); // 9
  console log(negyzet()); // NaN
  console.log(negyzet("Micimackó")); // Nan
5
```



```
parameter2.js
```

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

Régi módszer hiányzó paraméterek kezelésére (hatvanyAlapertelmezettRegi, js)

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

Alapértelmezett paraméter érték (hatvanyAlapertelmezett.js)

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

- 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

```
const osszead = (a, b) \Rightarrow a + b;
  const muvelet = function(a, op, b) {
    console.log('\{a\} + \{b\} = \{\{op(a, b)\}'\};
4
  muvelet (3. osszead. 5): //3 + 5 = 8
```

```
const muvelet = function(a, op, b) {
     console.log('\{a\} + \{b\} = \{op(a, b)\}');
3
   muvelet (3, (a, b) \Rightarrow a + b, 5); // 3 + 5 = 8
   muvelet (4,
6
     function(a, b) {
        return a + b:
8
9
   ): // 4 + 7 = 11
10
```

Függvények definiálása és azonnali hívása (paramFv3.js)

```
(function(a, op, b) {
    console \log( \$\{a\} + \$\{b\} = \$\{op(a, b)\} );
  \{(3, (a, b) \Rightarrow a + b, 5): // 3 + 5 = 8\}
4
  ((a, op, b) => \{
    console.log('\{a\} + \{b\} = \{op(a, b)\}');
 \{(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

Környezet (environment)

- adott pillanatban létező változók és értékeik
- gvakorlatilag soha nincs üres körnvezet



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

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

Rekurzív hatványozás (rekurzio.js)

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

Fibonacci-számok (fibonacci.js)

Fibonacci-sorozat: másodrendben rekurzív sorozat. Képzeletbeli 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!



Négyzetgyökvonás (gyok.js)

Készítse el a gyok függyényt, mely Newton módszerrel meghatározza és visszatérési értékként 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

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

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:

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$
 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.

Legnagyobb közös osztó (Inko.is)

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



Objektumok

- tulajdonság (kulcs) érték párok (csak a null-nak és az undefined-nak nincsenek tulaidonságai a nyelvben)
- minden tulajdonság egyedi az objektumban
- a tulajdonság lehet adat vagy függvény (metódus)
- a tulaidonságot az értéktől : választia el, a párokat egymástól .

Obiektum definiálása literálként

```
const hg = {
    nev: "Kovács...lstván".
    neptun: "a1b2c3",
    zh: 12
5
```

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

```
console.log(hg.zh); // 12
   hg.zh = 14; // Tulajdonságok változtathatók
   console.log(hg.zh); // 14
   // De const miatt az objektum nem váltható le
10
   hg = { // TypeError: invalid assignment to const 'hg'
12
   nev: "Nagy Péter",
13
   neptun: "1a2w3e".
14
   zh: 13
16
```

Két kötés (referencia) ugyanarra az objektumra

```
// Kötések (binding), nem klasszikus változók
   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 tulaidonságokon történő iterálásra

Ha a tulajdonság neve kötéssel adott, a . operátor nem használható \rightarrow objektum["tulajdonság"]

Tulaidonságok elérése

```
// Tulaidonság létezésének tesztelése
   console.log("nev" in hg); // true
   console.log("evfolyam" in hg); // false
25
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]);
32
```

Objektumok tartalmának másolása:

Object.assign(cél, forrás1, forrás2, ..., forrásN) Visszatérési érték: cél

```
Tulajdonságok másolása
```

```
// Objektumok tartalmi másolása
   const hg2 = {
     nev: "Kovács...Emőke".
37
   zh2: 19
38
39
40
   let egyesitett = Object.assign({}, hg, hg2);
   nvomtat(egvesitett):
   Object.assign(hg2, hg);
42
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 törölhetőek

```
Tulajdonságok hozzáadása, törlése
   // Tulajdonságok utólagos hozzáadása, elvétele
   hg.zh1 = hg.zh:
   delete hg.zh;
48
   hg.zh2 = 20;
   nvomtat(hg):
49
```

Kimenet

nev : Kovács István neptun : a1b2c3

zh1 : 15 zh2 : 20

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

```
Metódus hozzáadása
   let nev = "Fekete,,Péter";
    let neptun = "abcdef";
    let zh1 = 12:
    let zh2 = 8:
55
    const hg3 = {
      nev: nev.
      neptun: neptun,
      zh1: zh1.
      zh2 · zh2
60
61
    nyomtat(hg3);
    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

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

Metódus hozzáadása literálként

```
72
   const hg5 = {
     nev: "Mezei, Virág",
73
74
     neptun: "1qay2w",
75
     zh1: 14.
76
     zh2: 19,
77
     getAlairas: function() { // metódus literállal
78
        return (this zh1+this zh2) >= 20;
79
80
   console.log(hg5.getAlairas()); // true
81
```

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

```
let t1 = []; // \ddot{u}res t\ddot{o}mb
console.log(typeof t1); // object
let t2 = ["A|ma", "Banán", "Citrom"];
console.log(t2[1]); // Banán
t2[1] = "Burgonya";
console.log(t2[1]); // Burgonya
console.log(t2.length); // 3
```

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

```
Tömbök bejárása
    function nyomtat1(tomb) {
      for(let elem of tomb) {
10
11
        console.log(elem):
13
14
   nyomtat1(t2);
15
16
   function nyomtat2(tomb) {
17
      for(let idx in tomb) {
18
        console.log(idx, ":", tomb[idx]);
19
20
   nvomtat2(t2);
21
```

```
Kimenet
Alma
Burgonya
Citrom
0 : Alma
  : Burgonya
  : Citrom
```

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

Objektum tulajdonságainak visszaadása tömbként

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

Kimenet

```
: egy
 : ketto
2 : harom
```

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

```
Tömbök elemei
   t2[3] = "Dió":
30
31
   nvomtat2(t2):
   t2[5] = "Füge";
33
   console. log(t2.length); // 6
   console.log(t2[4]); // undefined
34
   nyomtat2(t2);
```

Kimenet

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

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

Hiányos tömbök

```
let t3 = ["Alma", "Citrom", ];
37
   console.log(t3.length); // 3
   nyomtat2(t3);
38
   let t4 = ["Alma", "Citrom", undefined];
39
   console.log(t4.length); // 4
40
   nvomtat2(t4):
41
```

Kimenet

```
: Alma
```

2 : Citrom

: Alma

2 : Citrom

: undefined

Tömbelem törlése: delete operátorral

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

Kimenet

: Alma

: undefined

Tömbelem törlése

Verem műveletek

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

```
Kimenet 1/2
                                                                                      Kimenet 2/2
    let t5 = [1, 2, 3];
46
47
    t5 push (4);
48
    nvomtat2(t5):
    console \log(t5 \text{ pop}()); // 4
49
50
    nvomtat2(t5):
51
    t5. unshift (0):
52
    nyomtat2(t5);
53
    console log(t5.shift()); // 0
    nvomtat2(t5):
```

Tömbök egyesítése: concat()

Tömbök egyesítése

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

Kimenet

```
O : Alma
1 : Banán
2 : 1
3 : 2
```

4:3

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

- \blacksquare 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

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

Új tömb létrehozása meglévő tömb elemeinek kimásolásával

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

Úi tömb létrehozása meglévő alapján

```
let t10 = ["Alma", "Banán", "Citrom", "Dió"];
68
69
70
   console.log(t10.slice(1, 2)); // ["Banán"]
71
   console log(t10); // ["Alma", "Banán", "Citrom", "Dió"]
72
73
   console. \log(t10.s|ice(-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 = ["A|ma", "Banán", "Citrom", "A|ma"];
76
77
    console log(t11 indexOf("Banán")); // 1
79
    console \log (t11 \text{ indexOf}("\text{Dió"})); //-1
    console.log(t11.lastIndexOf("Alma")): // 3
80
81
    console log(t11 indexOf("Alma", 1)); // 3
82
    console \log(t11) indexOf("Alma", -3)): // 3
```

Rendezés:

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

```
Rendezés különféle sorrendekbe
```

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

- forEach() minden egyes elemmel külön meghívja a paraméter fv.-t
- every() Igazat ad, ha a tesztelő fv. minden egyes elemre igazat ad → logikai és
- \blacksquare some() Igazat ad, ha a tesztelő fv. legalább egy elemre igazat ad \rightarrow 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



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

Többdimenziós tömb egydimenziós tömbök egymásba ágyazásával hozható létre

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

```
Kimenet
0.0:Alma
```

```
0 1 : Banán
10:1
11:2
12:3
```

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

$$\left[\begin{array}{cc} a & b \\ c & d \end{array}\right] = a \times d - b \times c$$

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

Dekompozíció

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

Hasonló dekompozíció objektumokkal is megvalósítható.

Dekompozíció

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

```
function at lag1() {
    let osszeg = 0:
3
    for(let adat of arguments) {
      osszeg += adat;
5
6
    return osszeg / arguments.length;
  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

```
function at lag2 (...adatok) {
10
     let osszeg = 0;
11
     for(let adat of adatok) {
12
13
       osszeg += adat:
14
15
     return osszeg / adatok.length:
16
   console.log(atlag2(1, 2, 3, 4)): // 2.5
17
```

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

Független változók ↔ tömb

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

Néhány apróság

A függvények paraméter átadásának módja

- alapvetően érték szerinti (pass by value), de
- lacktriangle az objektumoknak a referenciáját adja át, de azt érték szerint (pass by sharing) ightarrowaz objektum csak a fv.-en belül cserélhető le, de az eredeti objektum meglévő tulajdonságainak módosítása látszik a hívás után is

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

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

```
let a = 1:
   // let a = 2; SyntaxError: redeclaration of let a
    const b = 1:
   // const b = 2; SyntaxError: redeclaration of const b
    var c = 1:
    var c = 2:
    console log(c); // 2
8
    const muvelet = (n, m) \Rightarrow n + m;
    //const muvelet = (n, m) \Rightarrow 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
```

Objektumok és tömbök

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

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

Typeof működése különböző típusú adatokkal \rightarrow pl. fv. létezésének ellenőrzésére

```
console.log(typeof(console.log)): // function
  console.log(typeof({kulcs: "ertek"})); // object
  console \log(typeof([1, 2, 3])); // object
3
  console.log(typeof(null)); // object
  console.log(typeof(undefined)); // undefined
5
6
  console.log(typeof(true)); // boolean
  console.log(typeof(1)); // number
8
  console.log(typeof("szo")): // 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 (hystack.js)

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

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

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

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