

JavaScript



Szerkesztette: Kiss Gábor

PG 2024

Tartalomjegyzék

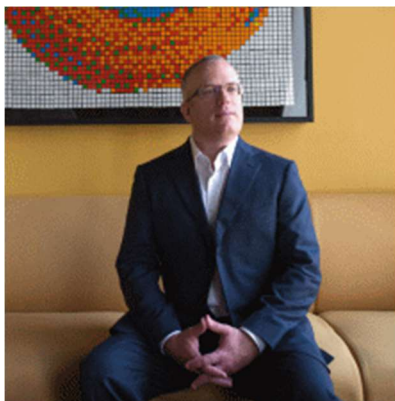
1	Bevezetés.....	6
2	A fejlesztési környezet kialakítása	7
3	JavaScript programozási alapok.....	8
3.1	JavaScript elhelyezése a HTML kódba	8
3.2	Variációk a "Hello világ!" programra.....	8
3.2.1	Írás a webböngésző konzoljára	8
3.2.2	Írás a HTML dokumentumba	9
3.2.3	Írás egy HTML elembe	9
3.2.4	Írás figyelmeztető ablakba	9
3.3	Szintaxis.....	9
4	Változók	11
4.1	Változók megadása.....	11
4.2	Típusok	11
4.3	Operátorok	12
4.3.1	Értékadó operátor	12
4.3.2	Összehasonlító operátor	12
4.3.3	JavaScript logikai operátorok.....	13
4.3.4	Aritmetikai operátor	13
4.3.5	Szövegösszefűző operátor	14
4.3.6	Bitenkénti operátor	14
4.3.7	Feltételes operátor	15
4.3.8	Unáris operátor	15
4.3.9	Kapcsolati operátor	15
4.3.10	Operátorok precedenciája.....	15
4.4	Matematikai függvények	16
4.5	Típuskonverzió	17
4.6	Hatókör	17
5	Adatbevitel, véletlenszámok	20
5.1	Adat bekérése	20
5.2	Véletlenszámok	20
5.2.1	A <code>Math.random()</code> függvény.....	20
5.2.2	1-nél nagyobb számok generálása.....	20
5.2.3	Egész számok	20
5.2.4	Egész számok generálása tetszőleges intervallumon	21
5.3	Feladatok	21
6	Feltételkezelés	22
6.1	<code>if ... else</code>	22
6.2	<code>switch</code>	22
6.3	Feladatok	23
7	Ciklusok	24
7.1	<code>for</code>	24
7.2	<code>for ... in ...</code>	24
7.3	<code>for ... of ...</code>	25
7.4	<code>while</code>	26
7.5	<code>do ... while</code>	26
7.6	Ugró utasítások.....	26
7.7	Feladatok	27
8	Stringek kezelése.....	28

8.1	String létrehozása JavaScript-ben.....	28
8.2	Escape karakterek	28
8.3	A String metódusai és tulajdonságai	28
8.4	Feladatok	31
9	Tömbök	32
9.1	Mik azok a tömbök?	32
9.2	Mire való a tömb?.....	32
9.3	Mi az az index?.....	32
9.4	Tömb létrehozása.....	33
9.5	Tömb elemének elérése	33
9.6	Tömb összes elemének elérése	34
9.7	Tömb elemének módosítása	34
9.8	Új elem hozzáadása a tömbhöz	34
9.9	Tömb hosszának megállapítása	35
9.10	Tömbelem törlése.....	35
9.11	Asszociatív tömb.....	35
9.12	Többdimenziós tömbök	35
9.13	Tömbök metódusai.....	36
9.14	Feladatok.....	40
10	Halmazok	41
10.1	A halmazok tulajdonságai, metódusai	41
10.2	Feladatok.....	42
11	A Map adatszerkezet	43
11.1	Feladatok.....	44
12	Függvények	45
12.1	Függvények létrehozása deklarációval	45
12.2	Függvény kifejezések.....	45
12.3	Nyíl függvények.....	46
12.4	A <code>Function</code> konstruktor	46
12.5	Önmagát hívó függvények.....	46
12.6	Paraméterek átadása.....	47
12.7	Hoisting.....	49
12.8	Callback függvények	49
12.9	Visszaadott függvények	50
12.10	Closure	51
12.11	Generátorok.....	51
12.12	Rekurzió.....	52
12.13	Feladatok.....	53
13	Objektumorientáltság	54
13.1	Objektumok létrehozása.....	54
13.2	Az <code>Object</code> osztály	54
13.3	Tulajdonságok.....	54
13.4	Metódusok.....	55
13.5	Objektumok megjelenítése.....	55
13.6	Lekérdezők és beállítók	56
13.7	A függvények, mint objektumok	57
13.8	A <code>this</code> jelentése.....	57
13.8.1	<code>call()</code> és <code>apply()</code>	57
13.8.2	A <code>bind()</code> metódus.....	58
13.9	Konstruktorok	59

13.10	Prototípus	59
13.11	Prototípus lánc	59
13.12	Öröklődés.....	60
13.12.1	Object.create()	60
13.13	Osztályok	61
13.14	Objektumszintű metódusok	61
13.15	Feladatok.....	62
14	Reguláris kifejezések.....	64
14.1	Reguláris kifejezések létrehozása	64
14.1.1	Létrehozás	64
14.1.2	Egyszerű minta	64
14.1.3	Speciális karaktere.....	64
14.1.4	Kapcsolók.....	66
14.2	Reguláris kifejezések használata.....	66
14.2.1	A test() metódus	66
14.2.2	Az exec() metódus	66
14.2.3	A match() metódus.....	66
14.2.4	A replace() metódus.....	66
14.2.5	A search() metódus	67
14.2.6	A split() metódus.....	67
14.3	Zárójelezett szövegrész megfeleltetése.....	67
14.4	Példák a RegEx alkalmazására	67
14.4.1	Dátum feldolgozás.....	67
14.4.2	Bankszámlaszám ellenőrzése	68
14.5	Feladatok.....	69
15	Kivételkezelés	70
16	Destrukurálás.....	71
17	Eseménykezelés	73
17.1	A JavaScript eseményei	73
17.2	Eseménykezelés az addEventListener() metódussal	75
17.3	Feladatok.....	78
18	DOM manipuláció.....	79
18.1	A HTML DOM	79
18.1.1	HTML elemek DOM-beli viszonyai	80
18.1.2	A DOM gyakorlati jelentősége	80
18.2	JavaScript DOM-műveletek, egy példán keresztül.....	81
18.2.1	Objektumok megkeresése a DOM-fában	81
18.2.2	Elemek beszúrása és módosítása.....	82
18.2.3	Objektumok törlése	85
18.3	Stíluselemek megváltoztatása	87
18.4	Feladatok.....	88
19	Űrlapok használata	91
19.1	Feladatok.....	92
20	Aszinkron JavaScript.....	93
20.1	Szinkron futás	93
20.2	Aszinkron futás	93
20.3	Callback Hell	93
20.4	Promise	95
20.4.1	Promise állapotok.....	96
20.4.2	Promise létrehozása.....	96

20.4.3	Az executor függvény	96
20.4.4	Válasz a szervertől: a <code>then</code> és a <code>resolve</code> függvény	96
20.4.5	Hibakezelés: <code>catch</code> és <code>rejected</code>	97
20.4.6	Láncolt Promise-ok	98
20.5	Az <code>async</code> és az <code>await</code>	99
20.6	Feladatok	100
21	Lokális tárolók (Storage API)	101
21.1	Feladatok	102
22	JSON	103
22.1	Biztonsági problémák	104
22.2	JSON szöveg konvertálása JavaScript objektummá	104
22.3	Feladatok	105
23	A REST API architektúra	106
23.1	Mi az a REST?	106
23.2	Mit kell tartalmaznia egy REST API kliens kérésnek?	107
23.2.1	Egyedi erőforrás azonosító	107
23.2.2	Metódus	107
23.2.3	HTTP fejlécek	108
23.2.4	Adat	108
23.2.5	Paraméterek	108
23.3	Mit kell tartalmaznia egy REST API szerver válasznak?	109
23.3.1	Állapotsor	109
23.3.2	Fejlécek	109
24	AJAX	110
24.1	Az XMLHttpRequest objektum	110
24.2	Feladatok	113
25	Fetch API	114
25.1	Kérés küldése	114
25.2	Kérés megszakítása	115
25.3	A <code>fetch()</code> használata CORS-sel és hitelesítő adatokkal	116
25.4	JSON adatok küldése	117
25.5	A <code>Request</code> és a <code>Header</code> objektumok	117
25.6	Feladatok	118
26	Modulok	119
27	Mellékletek	120
28	Források:	124

1 Bevezetés



A JavaScript 1995-ben, az akkor a Netscape-nél mérnökként dolgozó **Brendan Eich** alkotásaként indult hosszú, hódító útjára. Ennek megfelelően elsőként a korszak meghatározó jelentőségű webböngészőjében, a Netscape Navigatorban jelent meg a nyelv támogatása, méghozzá rögtön 1996 elején, a program 2.0-s verziójában. Eredetileg **Mocha**-nak, kicsit később **LiveScript**-nek nevezték volna, de mivel akkoriban a Netscape épp szoros üzleti kapcsolatban állt a Java-t birtokló Sunnal, egy meglehetősen elhibázott marketing-döntés nyomán úgy vélték, hogy az akkor már egészen sikeres Java népszerűségét meglovagolják, és végül a nyelv a **JavaScript** nevet kapta. Neveik alapján arra számíthatnánk, hogy a két nyelvben számos párhuzamra kellene

akadnunk, ám valójában azok csak kis mértékben hasonlítanak. Valóban voltak a fejlesztés első, sürgetett kezdeti szakaszában a Netscape-nél olyan felsőbb szintű igények, hogy a nyelv hasonlítson a Java-hoz, de ez ezen a szinten meg is állt. A programozási nyelv neve még napjainkban is rengeteg félreértést szül, így aztán a kissé furcsa névválasztás tekintetében a rövid és tömör lényeg: *A két programozási nyelvnek semmi köze sincs egymáshoz.*

Az erről szóló legelterjedtebb mondás a "Java is to JavaScript what Car is to Carpet.", amelynek magyar megfelelője akár így is szólhatna: "A Java-nak annyi köze van a JavaScript-hez, mint a rókának a parókához" (eredetileg autónak a szőnyeghez).

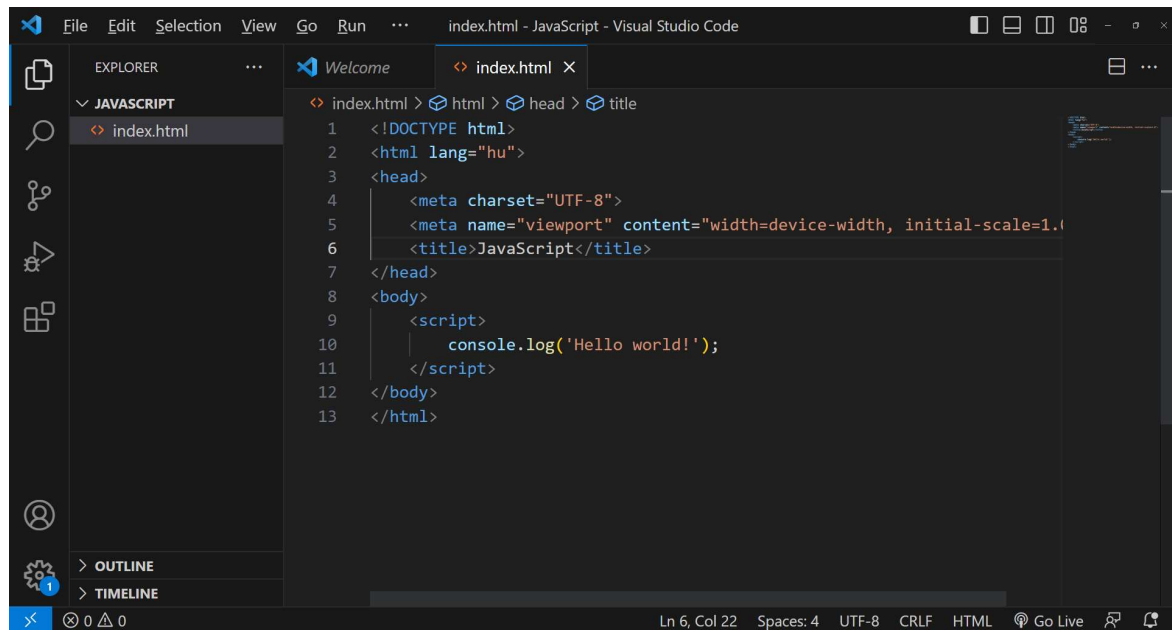
Megjelenését követően néhány hónappal később a Microsoft is beépítette saját JavaScript megvalósítását, a **JScript**-et az Internet Explorer 3-ba, majd újabb hónapok elteltével a Netscape a JavaScript szabványosítását elősegítendő, elküldte azt az **ECMA International**-nak (*European Computer Manufacturers Association*) és még 1997 júniusában megjelent az **ECMAScript** szabvány első kiadása. A szabványon tovább dolgoztak, és az első kiadást pontosan egy évvel követte a második, az ISO/IEC 16262 nemzetközi szabvánnyal való összhang megerősítése érdekében.

A szabvány 1999 decemberében, a 3. kiadásában érte el első igazán jelentős mérföldkövét, és attól kezdve tekinthetjük a nyelvet igazán stabilnak, megbízhatónak. A 4. verzió soha sem érte meg a végleges kiadását számos eltúlzott újítása, összetettsége miatt, így 2008-ban végleg elhagyatottak (*abandoned*) nyilvánították. Az 5. verzió 2009 decemberében jelent meg olyan fontos újításokkal, mint a `foreach`, a `map` és a `filter`. 2015 júniusában jelent meg az azóta is leginkább meghatározó **ECMAScript 6**. Újítása azóta is meghatározóak és széles körben alkalmazottak. A legismertebbek ezek közül a `let`, a `const`, a `for...of` ciklus, a Python-stílusú generátorok, a függvények innentől kezdve alapértelmezett paramétereket is kaphatnak és egy egészen új módon, az arrow function expression módszerrel (`() => {...}`) is definiálhatóak. A 2016-os 7. kiadás nem hozott jelentős újdonságokat, ám a 2017-ben megjelent **ECMAScript 8**-ban a nyelv újabb fejlesztéseket kapott elsősorban a párhuzamosan futtatható programszálak területén az *Async* függvények és a *Shared memory and atomics* bevezetésével.

A fentiek alapján kijelenthető-e akár az is, hogy a továbbiakban ECMAScript programozásról fogunk olvasni, abban fogunk programozni? A kérdésnek van alapja, hiszen eredetileg az ECMAScript főleg a JavaScript-ből vált szabvánnyá, majd onnantól kezdve a JavaScript az ECMAScript szabványa szerint fejlődik, mégis az egyik csak egy szabvány, míg a másik a szabványból származó programozási nyelvi megvalósítás. Ugyanez az ECMAScript a közös szabványa a JScript, ActionScript, TypeScript stb. nyelveknek is, azonban a JavaScript az ECMAScript kétségkívül legnépszerűbb implementációja. A helyes megfogalmazás tehát valahogy így hangzik: a továbbiakban az ECMAScript 8 alapú JavaScript programozással fogunk foglalkozni.

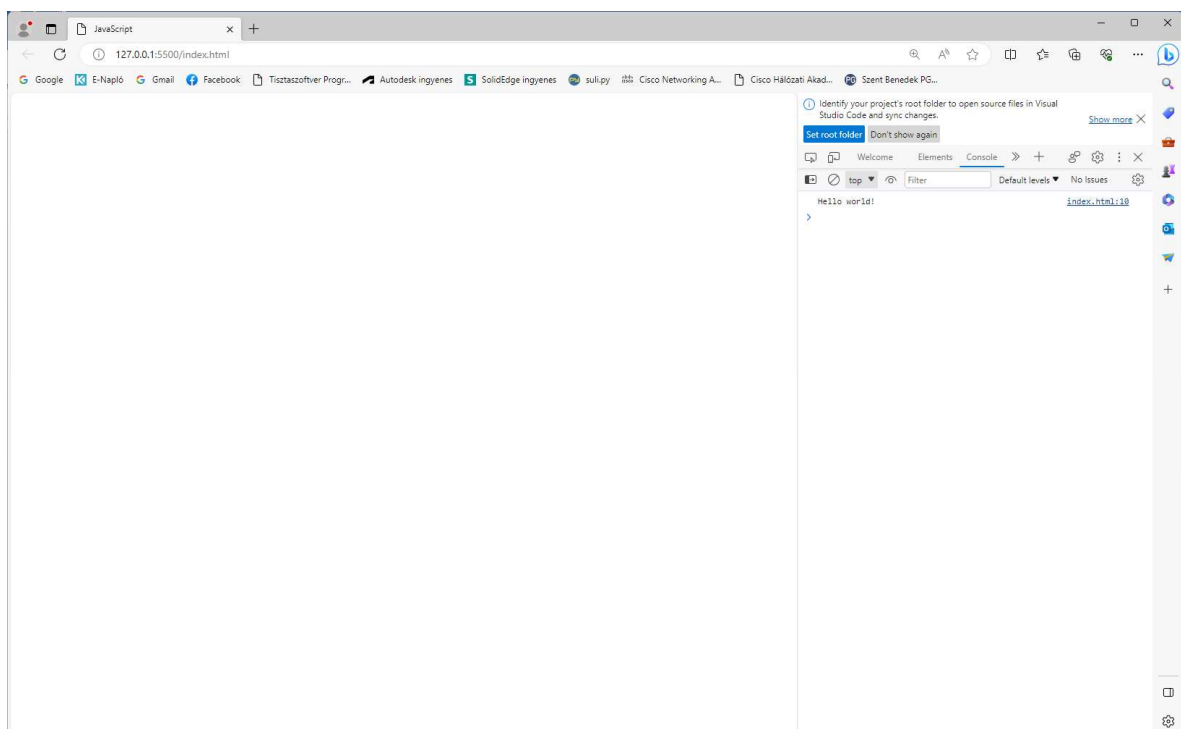
2 A fejlesztési környezet kialakítása

A JavaScript fejlesztéshez több szövegszerkesztő közül válogathatunk (pl.: Notepad, Notepad++, PS Pad...). Ha komolyabb tudással rendelkező fejlesztőkörnyezetet szeretnénk használni, akkor jó választás lehet a Visual Studio Code, amihez telepítsük a Live Server beépülő modult is.



Visual Studio Code

Az ablak jobb alsó sarkában a státuszsorban található „Go Live” feliratra kattintva megnyílik a Live Server, ahol megjelenik az aktuális weboldalunk. Az F12 billentyűvel megnyithatjuk a böngésző konzolját, ahol megjelennek a JavaScript programunk kimenetei és hibaüzenetei.



Live Server megnyitott „console”-al

3 JavaScript programozási alapok

Először nézzük meg, hogyan jeleníthetjük meg az üzeneteket JavaScriptben.

3.1 JavaScript elhelyezése a HTML kódba

Mivel a JavaScript futtatását a webböngésző végzi, a HTML kódba kell illesztenünk a scriptünket. Ezt kétféleképpen tehetjük meg.

Az első lehetőségként a `<script>...</script>` tag-ek közé illesztjük a JavaScript kódot. Készítsünk egy `index.html` nevű állományt az alábbi tartalommal:

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <script>
    console.log("Hello világ!");
  </script>
</body>
</html>
```

Ha kész, nyissuk meg egy webböngészőben, és az előző fejezetben már említett módon tekintsük meg az oldal által generált konzolkimenetet.

A JavaScript kódokat külön fájlban is elhelyezhetjük (pl.: `test.js`). Ebben az esetben az `index.html` állomány csak hivatkozást tartalmaz a beillesztett JavaScript fájlra:

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <script src="test.js"></script>
</body>
</html>
```

A hivatkozott `test.js` állomány tartalma (ugyanabban a könyvtárban, ahol az `index.html` is található):

```
console.log("Hello világ!");
```

A scripteket a `<head>` és a `<body>` tag-ek között is elhelyezhetjük.

3.2 Variációk a "Hello világ!" programra

Az elmaradhatatlan fejezet következik rögtön négy megoldással, amelyek közül egyelőre maradjunk a legelsőnél, de mindenképpen próbáljuk ki mindegyiket legalább egyszer.

3.2.1 Írás a webböngésző konzoljára

Ezzel a módszerrel a program futásakor mindenképpen létrejövő `console` objektum `log()` metódusát használjuk, amely az objektumorientált programozásban szokásos legelterjedtebb módon, egy pont karakterrel illeszthető egybe:

```
console.log();
```


3.2.2 Írás a HTML dokumentumba

Programjaink tesztelésére alkalmas a `document.write()` módszer. Közvetlenül a HTML dokumentumba ír. A kiírás formázására használhatók a HTML tag-ek is.

```
document.write('<b>Helló</b> világ!');
```

Ha a dokumentum kiírását követően futtatjuk, törli a dokumentum tartalmát. Ezért csak tesztelésre használjuk.

```
<h1>JavaScript</h1>
<p>Ez a tartalom törlődik, ha megnyomod a gombot.</p>
<button type="button" onclick="document.write('Hello világ!')">Próbáld ki</button>
```

A gomb működését az „Eseménykezelés” című fejezetben részletesen tárgyaljuk.

3.2.3 Írás egy HTML elembe

A HTML DOM (Document Object Model) alkalmazásával közvetlenül a HTML tartalmat módosíthatjuk.

```
<p id="teszt"></p>
<script>
    document.getElementById("teszt").innerHTML = "Hello világ!";
</script>
```

A HTML DOM használatával később részletesen foglalkozunk.

3.2.4 Írás figyelmeztető ablakba

A JavaScript lehetőséget ad arra, hogy üzenetünket egy *alert* ablakban jelenítsük meg.

```
window.alert("Hello világ!");
```

Mivel a window objektum minden objektum őse, nem kötelező kiírni. A program működik, ha az `alert()` módszert önmagában használjuk.

3.3 Szintaxis

A JavaScript legfontosabb alaptulajdonságai:

- Nem típusos nyelv, a változók típusát nem kell megadni.
- A típuskonverzió automatikus.
- Az utasításokat pontosvesszővel (;) zárhatjuk le, de nem kötelező a használata.
- Főprogram külön nincs; a végrehajtó fentről lefelé sorban végrehajtja az utasításokat.
- Egy soros megjegyzést a `//`, több sorost pedig a `/* ... */` struktúrával tudunk létrehozni.
- A blokkokat kapcsos zárójelek közé tesszük: `{...}`.
- A műveletek, az értékadás és az összehasonlítás a legtöbb nyelvben megszokott szintaxissal történik.
- A feltételkezeléssel és ciklusokkal kapcsolatos struktúrák szintaxisa szintén a Javában megszokott: `if ... else`, `switch ... case ... default`, `for`, `while` stb.
- Ugró (`goto`) utasítás itt sincs, viszont `break` és `continue` van.

- Függvényeket a `function` kulcsszóval tudunk létrehozni. A paraméterek típusát és a visszatérési érték típusát nem kell megadni, egyebekben megegyezik a szintaxis a Java-ban megszokottal (pl. `function add(a, b) {return a + b}`).
- Létezik kivételkezelés, melynek szintaxisa a Java-hoz hasonló: `try ... catch ... finally`, ill. van hozzá tartozó `throw` utasítás. De itt nincs szigorú kivétel hierarchia, valamint nem kell (nem is lehet) megadni azt, hogy melyik függvény milyen kivételt vált ki.
- A JavaScript funkcionális nyelv is, és lambda kifejezéseket is használhatunk. Ezeket nyíl függvényeknek (*arrow function*) hívjuk, pl. `(a, b) => a + b`.
- Objektumorientált lehetőségeket is tartalmaz.
- Elérhetők a legfontosabb alap adatszerkezetek: lista, halmaz, asszociatív tömb, számos hasznos művelettel.
- Gazdag a matematika, a string és a dátum kezelés könyvtára is.
- Tetszőleges HTML elemet képes létrehozni, megváltoztatni, törölni, számos keresési lehetőséggel.
- Case Sensitive nyelv. A kis és nagybetűk különbözőek. Pl.: `valtozo` és a `Valtozo` különböző azonosítók.
- Az azonosítók tartalmazhatnak betűket, számokat aláhúzás (`_`) és dollár (`$`) jelet, de számmal nem kezdődhetnek.
- A nyelv foglalt szavai (pl. utasítások) nem használhatók azonosítóként.

4 Változók

A JavaScript nem típusos nyelv: a változóknak nem kell (és nem is tudjuk) megadni a típusát.

4.1 Változók megadása

A változókat deklarálni kell a használat előtt, például az alábbi módon:

```
var a = 2;  
console.log(a);
```

A változóknak alapvetően kétféle megadási módjuk van:

- A fenti példában is látható `var` kulcsszóval. Ez volt a kezdeti megadási módszer, de mára idejétmúlttá vált, használatát célszerű kerülni.
- A 2015-ben megjelent ES6-os szabványban bevezették a `let` és a `const` megadási módokat, a változók és a konstansok számára. A `const` kulcsszóval megadott "változó" értéke később nem módosítható. Az új megadási mód bevezetésének az oka kettős: egyrészt megjelent a blokk hatókör (*scope*), másrészt a funkcionális programozás; mindkettőről lesz szó a későbbiekben.

```
let b = 3;  
b = 4;  
const c = 5;  
c = 6; //hiba
```

Megjegyzés: a `var` változók újradefiniálhatók. Az alábbi tehát helyes:

```
var x = 5;  
console.log(x);  
var x = "alma";  
console.log(x);
```

Ez ellentmond a más programozási nyelvekben megszokottaknak. A `let` esetében az újradefiniálás hibát ír ki.

4.2 Típusok

A típusokat tehát nem adjuk meg, de ettől függetlenül a változóknak vannak típusai. Az alábbi típusokat különböztetjük meg:

- **number** (szám): ez alatt tetszőleges számot értünk, tehát egészet, tizedes törtet, negatív, pozitív. A szokásos szám megadási módszerek (pl. `123`, `-14.2`, `5e5`, `0xFA055`, `0b10101010` stb.) működnek. A JavaScript minden number típusú változót lebegőpontosan tárol.
- **boolean** (logikai): az igaz és hamis értéket a `true` ill. `false` kulcsszavakkal adhatjuk meg. Automatikus konverzió történik a többi adattípusról, pl. a `0`, az üres string vagy az üres objektum a logikai hamis, a többi a logikai igaz értéknek felel meg.

- **string** (szöveg): megadhatjuk egyszeres és kétszeres idézőjelek között is: a `"hello world"` és a `'hello world'` egyenértékű. Az ES6-ban bevezetésre került egy újabb fajta megadás, melynek neve sablon szöveg (*template string*): ezt az egyszeres visszafelé idézőjelek közé ``hello world`` kell tenni, és akkor behelyettesíti a változók értékét, pl. ``Hello, ${name}!`` (feltesszük, hogy van egy `name` változó, pl. `let name = 'Csaba'`; ez esetben az eredmény `Hello, Csaba!` lesz). Stringeket összefűzni a `+` jellel tudunk, pl. `"Hello, " + "world!"`.
- **object** (objektum): Minden objektum, ami nem primitív és nem függvény. A `null` a nem létező objektum.
- **function** (függvény): ezeket külön típusként kezeli a JavaScript.
- **undefined** (nem definiált): ha egy változót csak deklaráltunk, de nem adunk neki értéket, vagy még nem is deklarálunk, annak a típusa ez.

Pár példa:

```
console.log(typeof(2));           // number
console.log(typeof(true));        // boolean
console.log(typeof('Aladár'));    // string
console.log(typeof({name:'Béla', age: 42})); // object
console.log(typeof(function f(a, b) {return a + b})); // function
console.log(typeof(nonexisting));  // undefined
```

4.3 Operátorok

Az **operandus** az informatikában nem más, mint egy kifejezés tagja, amelyen valamiféle módosítást hajtunk végre, vagy felhasználjuk egy másik operandus értékének módosításához.

Pl.: `x=5+8` kifejezésben az 5 és a 8 operandus, míg a `+` jel maga az operátor. A visszatérési értéke (matematikában az eredménye) 13, vagyis `x` értéke 13 lesz.

4.3.1 Értékadó operátor

Az értékadó operátor hozzárendel az operandusához egy adott típusú értéket. Ilyen operátor az egyenlőségjel (`=`). A bal oldali változóhoz hozzárendeli a jobboldali értéket.

```
let x=1;
let y;
x=y;
```

4.3.2 Összehasonlító operátor

Az összehasonlító operátor összehasonlít két értéket. Az összehasonlítandó értékek lehetnek számok, objektumok, logikai értékek vagy stringek.

A máshol szokásos összehasonlító operátorok itt is jelen vannak: `==` (egyenlőség vizsgálata), `!=` (nem egyenlő), `<`, `>`, `<=`, `>=`; mindegyik eredménye logikai igaz vagy hamis. A JavaScript speciális ebből a szempontból is. Mivel a típuskonverzió automatikus, a következő eredménye igaz lesz:

```
console.log(5 == "5"); // true
```

Ha azt szeretnénk, hogy csak akkor legyen az összehasonlítás eredménye igaz, ha a típusaik is megegyeznek, és az eltérő típusú értékek összehasonlításának eredménye minden esetben hamis legyen, akkor az `===` operátort használhatjuk (melynek tagadó párja a `!==`):

```
console.log(5 === "5"); // false
console.log(5 === 5);   // true
console.log(5 !== "5"); // true
console.log(5 !== 5);   // false
```

4.3.3 JavaScript logikai operátorok

A logikai operátorokat elsősorban feltételek írásakor szoktuk használni. Arra valók, hogy több feltételt fűzzünk össze, azaz megtehetjük, hogy olyan elágazást írunk, amely csak akkor halad tovább, ha az összes feltételünk igaz. Például, csak akkor engedjük be a felhasználót a weboldalunkra, ha elmúlt 18 éves és be van jelentkezve.

A tagadás (negáció) vagy logikai nem művelet, akkor ad igaz értéket, ha az állítás hamis. A `!(a < 5)` kifejezés, akkor igaz, ha az `a` változó értéke nem kisebb 5-nél.

A logikai és (and) művelet igaz eredményt ad, ha mindkét oldala igaz értékű. Az `a > 5 && a < 20` kifejezés igaz, ha `a` 5-nél nagyobb és 20-nál kisebb. Ha kettőnél több feltétel szerepel a kifejezésben `&&` operátorral összekötve, úgy mindegyik teljesülése esetén lesz igaz a kifejezés értéke.

A logikai vagy (or) művelet akkor ad igaz eredményt, ha legalább az egyik oldala igaz. Az `a == 3 || a == 5` kifejezés eredménye igaz, ha az `a` értéke 3 vagy 5. Ha `||` operátort kettőnél több kifejezés összekötésére használunk, igaz értéket kapunk bármely kifejezés teljesülésekor.

A	!A
Igaz	Hamis
Hamis	Igaz

A	B	A&&B
Hamis	Hamis	Hamis
Hamis	Igaz	Hamis
Igaz	Hamis	Hamis
Igaz	Igaz	Igaz

A	B	A B
Hamis	Hamis	Hamis
Hamis	Igaz	Igaz
Igaz	Hamis	Igaz
Igaz	Igaz	Igaz

Logikai operátorok igazságtáblázatai

4.3.4 Aritmetikai operátor

Az aritmetikai operátor az operandusok számértékeire hatva visszatér egy másik számértékkel. Ide tartozik az összeadás (+), a kivonás (-), a szorzás (*), az osztás (/) és a modulo vagy maradékosztás (%).

```
let z;
z=3+4; // összeadás
z=3-4; // kivonás
z=2*3; // szorzás
z=2/3; // osztás
z=3%2; // modulo
z=3**2; // hatványképzés (hatványalap**hatványkitevő)
```

```
let x,y;
x=5;
y=++x; // preinkrement
console.log(`x=${x}, y=${y}`);
x=5;
y=x++; // posztinkrement
console.log(`x=${x}, y=${y}`);
x=5;
y--x; // predekrement
console.log(`x=${x}, y=${y}`);
x=5;
y=x--; // posztdekrement
console.log(`x=${x}, y=${y}`);
```

Az `x += 5` (és társai) jelentése `x = x + 5`.

Az `x++` jelentése `x = x + 1`. Ha az eredményt értékül adjuk, akkor különbséget kell tenni az alábbi kettő között: `y = x++` (az `y` értéke az `x` eredeti értéke lesz, majd az `x` értéke eggyel növekszik), és `y = ++x` (az `x` értéke eggyel nő, majd az új értéket kapja meg az `y`).

4.3.5 Szövegösszefűző operátor

Egymástól elkülönített szövegeket, vagy egy változó ill. konstansnév utáni szöveget a `+` operátorral tudunk kiíratáskor összefüggővé tenni az alábbi módokon:

```
console.log('Addig jár a korsó a kútra, '+' amíg be nem vezetik a vizet.');
```

Sőt, akár még így is használható:

```
let x = 'kocka';
x += 'has'; // x értéke így: 'kockahas'
```

4.3.6 Bitenkénti operátor

A bitenkénti operátorok bit szintjén végzik el a számításokat. A bitenkénti tagadás: `~` művelet a kapott operandus bitjeit negálja. A bitenkénti és: `&`, vagy: `|`, kizáró vagy: `^` műveletek az operandusként kapott két érték közötti minden bitre elvégzik a műveletet.

```
let x = 0b01110101; //117
let y = 0b11010011; //211
console.log(~x); // -118
console.log(x & y); //81
console.log(x | y); //274
console.log(x ^ y); //166
```

```
Pl.:      01110101 -> 117
      & 11010011 -> 211
      -----
      01010001 -> 81
```

A bitek balra mozgatása: `<<`, bitek előjeles jobbra mozgatása: `>>`, bitek jobbra mozgatása, minden esetben balról 0-lal feltöltve: `>>>`.

```
console.log(5 << 1); //10
console.log(5 >> 1); //2
console.log(5 >>> 1); //2
```

A `<<` operátor az első operandusban megadott értéket bitenként balra tolja a második operandusban megadott értékkel. A jobb oldali helyeket 0-val tölti fel. Pl: `0101 << 1` eredménye 1010 lesz.

A `>>` operátor az első operandusban megadott értéket a második operandusban megadott számú bittel jobbra tolja. A jobb oldali bitek törlésre kerülnek. Az előjelbit nem kerül eltolásra, így a kapott érték előjele nem változik.

A `>>>` operátor az előzőhöz hasonlóan működik, de itt az előjelbitet nem veszi figyelembe.

4.3.7 Feltételes operátor

A JavaScript-ben is megtalálható a három tagú (*ternary*) operátor: `?:`. Az egyetlen operátor amelynek három operandusa van. A használatát egy példán illusztrálom: az `a = b > 2 ? c : d` jelentése a következő: ha `b` értéke nagyobb mint kettő, akkor `a` értéke legyen `c`, egyébként `d`. Ez egyébként csökkenti a kód olvashatóságát, így használatát jól meg kell fontolni. Megengedett akkor, ha paraméter átadásnál nem szeretnénk segédváltozót létrehozni, vagy egy `if` szerkezetben nem szeretnénk leírni kétszer ugyanazt, de ahol észszerű az `if ... else` szerkezet használata, ott érdemes azt használni.

4.3.8 Unáris operátor

Unáris operátoroknak azon operátorokat nevezzük, melyek csupán egyetlen operandussal rendelkeznek. A `delete` képes kitörölni objektumot, objektum tulajdonságot, vagy egy tömb egy adott értékét. A `typeof` visszatér egy stringgel, ami azt jelzi, hogy az operandusához (egy változó) milyen típusú érték van hozzárendelve. A `void` egy kifejezést definiál amit ki kell értékelni, visszatérési érték nélkül

4.3.9 Kapcsolati operátor

A kapcsolati operátorok összehasonlítják az operandusaikat és valamilyen logikai típusú értékkel (igaz vagy hamis) térnek vissza annak a függvényében, hogy az összehasonlítás feltétele miképp teljesül. Az `in` igaz értékkel tér vissza, ha egy objektum tartalmaz egy megadott nevű tulajdonságot. Az `instance of` igaz értékkel tér vissza, ha egy objektum neve (mint változó - az egyik operandus) olyan típussal rendelkezik, mint egy objektum típus (a másik operandus)

```
const gyumolcs = { fa: 'alma', fajta: 'Starking', szín: 'piros' };
console.log('fajta' in gyumolcs); //true
```

4.3.10 Operátorok precedenciája

Az operátorok precedenciája meghatározza a műveleti sorrendet, azaz, hogy melyik művelet hajtódjon végre hamarabb, és melyik később. Zárójelek használatával ezt a sorrendet felülírhatjuk. Egy listában felsoroltunk néhány fontosabb operátort a legnagyobb precedenciától a legkisebbig:

1. Negáció/növelés (`!`, `~`, `-`, `+`, `++`, `--`, `typeof`, `void`, `delete`)
2. Szorzás/osztás/maradékosztás (`*`, `/`, `%`)
3. Összeadás/kivonás (`+`, `-`)
4. Bitszintű eltolás (`<<`, `>>`, `>>>`)
5. Kapcsolat (`<`, `<=`, `>`, `>=`, `in`, `instanceof`)
6. Egyenlőség (`=`, `!=`, `===`, `!==`)

7. Bitszintű és (&)
8. Bitszintű kizáró vagy (^)
9. Bitszintű vagy (|)
10. Logikai és (&&)
11. Logikai vagy (||)
12. Feltétel (? :)
13. Értékadás (=, +=, -=, *=, /=, %=, <=, >=, >>=, &=, ^=)

4.4 Matematikai függvények

A Math objektum olyan matematikai állandókat és függvényeket tartalmaz, ami segíti a matematikai műveletek elvégzését. Az alábbi táblázatban a fontosabb állandók és függvények találhatók.

abs(x)	Visszaadja x abszolútértékét.
acos(x)	Visszaadja azt a szöget radiánban, amelynek a cosinusa: x.
asin(x)	Visszaadja azt a szöget radiánban, amelynek a sinusa: x.
atan(x)	Visszaadja azt a szöget radiánban, amelynek a tangense: x.
ceil(x)	Visszaadja x értékét a legközelebbi egész számra felkerekítve.
cos(x)	Visszaadja x cosinusát. (x értékét radiánban kell megadni).
E	Euler-féle szám. A természetes logaritmus alapja (megközelítőleg: 2,718).
exp(x)	Visszatérési értéke: E^x
floor(x)	Visszaadja x értékét a legközelebbi egész számra lefelé kerekítve.
log(x)	Visszaadja x természetes alapú logaritmusát.
log10(x)	Visszaadja x 10-es alapú logaritmusát.
max(x1,x2,..)	Visszaadja a paraméterben megadott értékek maximumát.
min(x1,x2,..)	Visszaadja a paraméterben megadott értékek minimumát.
PI	Visszaadja PI értékét 15 tizedes pontossággal (megközelítőleg: 3,14).
pow(x, y)	Visszaadja x^y értéket.
random()	0 és 1 közötti véletlenszámot ad vissza. (lásd később)
round(x)	X-et a legközelebbi egész számra kerekíti.
sin(x)	Visszaadja x sinusát. (x értékét radiánban kell megadni).
sqrt(x)	Visszaadja x négyzetgyökét.
tan(x)	Visszaadja x tangensét. (x értékét radiánban kell megadni).
trunc(x)	Visszaadja x egészrészét.

Nézzünk néhány példát az állandók és a függvények használatára és a valós számok kerekítésére.

```
console.log(Math.PI); //3.141592653589793
console.log(Math.abs(-23)); //23

console.log(Math.ceil(3.21)); //4
console.log(Math.ceil(-3.21)); //-3
console.log(Math.floor(3.67)); //3
console.log(Math.floor(-3.67)); //-4
console.log(Math.round(3.45)); //3
console.log(Math.round(3.67)); //4
console.log(Math.trunc(3.21)); //3
console.log(Math.trunc(-3.21)); //-3
```

4.5 Típuskonverzió

A JavaScript dinamikusan tipizált nyelv. Ez azt jelenti, hogy nem kell deklaráláskor meghatározni egy változó adattípusát, és az adattípusok automatikusan konvertálódnak, ahogy az a script futása során szükséges. Ha egy változót olyan helyen használunk, ahol a típusa nem megfelelő, a JavaScript automatikusan megpróbálja a megfelelő típusra konvertálni. A leggyakrabban a számok és a karakterláncok között fordul elő.

```
"dog" + "house" == "doghouse"; // összekapcsolta a két karakterláncot
"dog" + 4 == "dog4"; // a számot átalakította
4 + "4" == "44"; // karakterláncná
4 + 4 == 8; // összeadta a két számot
23 - "17" == 6; // a karakterláncot átalakította számmá
```

Amennyiben szeretnénk egy stringet számmá konvertálni használhatjuk a `parseInt()` (egészre konvertál) és a `parseFloat()` (valósra konvertál) metódusokat.

```
let a = "12";
let b = "23";
console.log(a + b); // "1223"
console.log(parseInt(a) + parseInt(b)); // 35

c = "12.34";
console.log(parseInt(c)); //12
console.log(parseFloat(c)); //12.34
```

Ha a `parseInt()` metódus paramétereként egy valós számot tartalmazó stringet, vagy valós számot adunk meg, úgy a konverzió eredménye a szám egészrésze lesz.

4.6 Hatókör

Angolul *scope*. Azt jelenti, hogy az adott változó hol érhető el. Háromféle hatókört különböztetünk meg:

- **Globális:** bárholnan elérhetőek. Ha egy változót "csak úgy" megadunk, `var`, `let` (vagy `const`) nélkül, akkor az automatikusan globális lesz, akárhol is hozzuk létre.
- **Függvény:** csak adott függvényen belül érhetőek el, de ott bárholnan, akárhol is hoztuk létre őket.
- **Blokk:** csak a saját blokkján belül érhetőek el.

Itt lényeges koncepcióbeli különbség van a `var` ill. a `let` és a `const` kulcsszavakkal létrehozott változók hatókörei között.

A `var` kulcsszóval létrehozott változók hatóköre csak globális és függvény lehet. Itt meg kell említeni az ún. hoisting mechanizmust: mindkét hatókör esetén, akárhol is hozzuk létre a változót, a deklarációt mindig felviszi az elejére: globális hatókör esetén a program tetejére, függvény hatókör esetén pedig a függvény fejléc alá; ez utóbbit még akkor is, ha egy belső blokkon belül történik a deklaráció. Ezt a kissé szokatlan megoldást két példán szemléltetem. Ennek megértése fontos amiatt, hogy megértsük a `let` és `const` létjogosultságát.

Ha a következőt írjuk:

```
console.log(a);  
var a = 2;
```

abból a szabvány szerint belül a következő lesz:

```
var a;  
console.log(a);  
a = 2;
```

Az eredmény: `undefined`, holott hibát várnánk.

Egy másik példa:

```
function f() {  
  if (3 > 2) {  
    var b = 3;  
  }  
  console.log(b);  
}  
f();
```

Azt várnánk, hogy ne írjon ki semmit, viszont kiírja a 3-at, ugyanis a fenti az alábbira alakul át:

```
function f() {  
  var b;  
  if (3 > 2) {  
    b = 3;  
  }  
  console.log(b);  
}  
f();
```

A probléma ezzel az, hogy szembemegy a más programozási nyelvekben (pl. Java-ban) megszokott logikával, ráadásul mivel nagyon valószínűtlen, hogy tényleg ezt szeretnénk írni, elnyelheti a hibákat.

A `let` (és `const`) kulcsszóval létrehozott változók (konstansok) hatóköre csak globális és blokk lehet. Ráadásul itt – a programozók által megszokott viselkedésnek megfelelően – nincs

hoisting. A `let` használatával mindkét fenti példa hibát jelezne – nagyon helyesen! E kulcsszavak bevezetésének megvan a létjogosultsága mert:

- A hatókörük sokkal természetesebb, mint a `var` esetén.
- Minden programozási nyelvben indokolt különbséget tenni a változók és konstansok között, ez utóbbi használatával a fordító, ill. futtató rendszerek optimalizálni tudják a memória használatot. A funkcionális programozási elemek bevezetése (pl. nyíl függvények (*arrow function*); ld. később) is indokoltá teszik a konstansok bevezetését.

Fontos viszont úgy tekintenünk erre, hogy egy alapvetően rosszul megtervezett megoldást *lecseréltek* egy jobbra. Helytelen lenne azt gondolni, hogy háromféle hatókör van a JavaScript-ben (noha a valóságban tényleg három van), ugyanis a régi és az új világban is kettő-kettő van. A `var` és a `let` + `const` megoldásokat nem szabad egyszerre használni: a már meglevő komponensek tovább fejlesztésekor, melyben `var`-t használtak (pl. mert a fejlesztése már 2015 előtt elkezdődött) továbbra is a `var` kulcsszó használata az indokolt, és ez esetben kerüljük a `let`-et és `const`-ot. Ugyanakkor egy teljesen új projekt esetén célszerű a `var`-t kerülni, és kizárólag a `let` ill. `const` kulcsszavakat használni.

Mivel a több milliárd működő weboldal és a több milliárd feltelepített böngésző miatt szinte lehetetlen megváltoztatni a JavaScript specifikációját (azt csak bővíteni lehet), egy rossz örökségként a `var` jó eséllyel örök időig benne marad. A web fejlesztők feladata annyira kikoptatni, amennyire csak lehet.

5 Adatbevitel, véletlenszámok

5.1 Adat bekérése

Adatokat legtöbbször űrlapokon keresztül tudunk megadni, amit egy JavaScript programmal feldolgozunk. Az űrlapok adatainak feldolgozásáról egy későbbi fejezetben lesz szó.

Lehetőség van felhasználói adatok bekérésére a `prompt()` metódus segítségével.

```
let a = prompt('Kérek egy számot: ', 0);
```

A `prompt()` metódus két paramétert vár. Az első az üzenet szövege, a másik egy alapértelmezett érték. A paraméterek megadása nem kötelező.

5.2 Véletlenszámok

Véletlenszámokra sok esetben szükségünk lehet a programozás során. Tiszta véletlenszámok előállítására a számítógépek nem képesek, de jó közelítéssel tudnak „véletlenszerű” számokat előállítani.

Nézzünk néhány lehetőséget, amikor véletlenszámok előállítására van szükség:

- szerencsejátékok (pl.: lottó, dobókocka...)
- sorsolás
- biztonsági kódok generálása

5.2.1 A `Math.random()` függvény

A `Math.random()` egy véletlenszámmal tér vissza 0 és 1 között, melyből a 0 zárt intervallum, azaz tartalmazhatja a generált szám, míg az 1 nyitott, így azt nem kaphatjuk eredményül. Azaz a kapott érték mindenképp 1 alatti lesz.

```
console.log(Math.random());
```

5.2.2 1-nél nagyobb számok generálása

Egynél nagyobb számok előállításához a matematikát kell segítségül hívunk:

```
console.log(Math.random()*10);
```

Ha $[0; 9]$ között szeretnénk számokat generálni, akkor elég a `Math.random()` függvény által visszaadott értéket tízzel megszoroznunk.

Ha 10-el szorzunk, akkor a generált szám maximum értéke 9 lehet. Ha egy 0-tól x -ig terjedő számot szeretnénk kapni, akkor annyi a dolgunk, hogy $x+1$ -el szorozzuk meg a `Math.random()` függvényét.

5.2.3 Egész számok

Ha egész számokra van szükségünk használjuk a `Math.floor()` függvényt, amely minden esetben lefelé fogja kerekíteni a kapott értéket..

Mivel mi egy véletlenszám értékét szeretnénk kerekíteni, így paraméterként a `Math.random()` függvényt kell megadnunk.

```
console.log(Math.floor(Math.random()*10));
```

A véletlenszámot megszorozzuk tízzel, majd kerekítjük, Fontos a sorrend, mert ha először kerekítünk és utána szorzunk, akkor minden esetben 0-át fogunk eredményül kapni.

5.2.4 Egész számok generálása tetszőleges intervallumon

Ez sem lehetetlen feladat, de ehhez is szükségünk lesz egy kis matematikára. Mindössze annyit teszünk, hogy eltoljuk az eredeti intervallumot az új kezdőértékre.

```
console.log(Math.floor(Math.random()*100)+100);
```

Alapvetően 0-tól 99-ig generáljuk a véletlenszámot, de mivel hozzáadunk 100-at, így 100 és 199 közötti számok kapunk eredményül.

5.3 Feladatok

1. Írj programot, ami beolvassa a felhasználó nevét, majd köszön neki!
2. Olvasd be egy téglalap két oldalának hosszát, majd írasd ki a kerületét és a területét!
3. Írj programot, ami beolvassa egy kör sugarát, majd kiírja a területét és a kerületét!
4. Írj programot, ami egy dobókocka dobást szimulál!
5. Írj programot, ami kihúz egy lottószámot!
6. Írj programot, ami bekér egy inch értéket és átszámolja cm-be! (1 inch = 2,54 cm)
7. Írj programot, ami bekéri az időt (óra, perc, másodperc) és kiszámolja, hogy a nap hányadik másodpercében vagyunk!
8. Írj programot, ami bekéri a dátumot (nap) és az órát, amiből kiszámolja, hogy a hónap hányadik órájában vagyunk!
9. Írj programot, ami bekér három számot, majd kiírja a számtani közepüket!
10. Írj programot, ami bekéri egy szög értékét fokban és átszámolja radiánba! ($360^\circ = 2\pi$ radián)
11. Kérd be egy intervallum alsó és felső értékét, majd generálj véletlenszámot az intervallumban!
12. Kérd be két pont koordinátáit és határozd meg a távolságukat!
13. Generálj egy „R” véletlenszámot 1 és 10 között, majd írasd ki az „R” sugarú kör kerületét és területét!
14. Generálj két véletlenszámot (A és B), majd írasd ki az A, B oldalú téglalap kerületét és területét!
15. Generálj egy valós típusú véletlenszámot és írasd ki 2 tizedes jegyre kerekítve!

6 Feltételkezelés

A programozási nyelvek egyik alapvető eleme az elágazás, más néven szelekció. A legtöbb programban előfordul, hogy valamilyen feltétel alapján döntenünk kell egy programrész végrehajtásáról. Ebben a fejezetben megnézzük, milyen lehetőségeink vannak erre a JavaScript programozási nyelvben.

6.1 if ... else

A legalapvetőbb feltételkezelő szerkezet az `if ... else`. A szintaxisa a legtöbb programozási nyelvben (pl. a Java-ban) megszokott, pl.:

```
let a = 5;
if (a > 0) {
  console.log("pozitív");
} else if (a == 0) {
  console.log("nulla");
} else {
  console.log("negatív");
}
```

6.2 switch

Ha az `if`-nek túl sok ága van, akkor érdemes az áttekinthetőbb `switch ... case ... default` szerkezetet használni. Ez a JavaScript-ben is megtalálható, a többi programozási nyelvben megszokott szintaxissal:

```
let nap = 4;
let napNev = "";
switch (nap) {
  case 1:
    napNev = "hétfő";
    break;
  case 2:
    napNev = "kedd";
    break;
  case 3:
    napNev = "szerda";
    break;
  case 4:
    napNev = "csütörtök";
    break;
  case 5:
    napNev = "péntek";
    break;
  case 6:
    napNev = "szombat";
    break;
  case 7:
    napNev = "vasárnap";
    break;
  default:
    napNev = "ismeretlen";
}
console.log(napNev);
```

6.3 Feladatok

1. Egy beolvasott számról döntse el a program, hogy -30 és 40 között van-e!
2. Két beolvasott szám közül írassuk ki a nagyobbikat! Azt is írassuk ki, ha egyenlők!
3. Egy beolvasott X számnak írjuk ki az előjelét (pozitív, negatív vagy nulla)!
4. Kérjünk be egy számot és döntsük el, hogy egész szám-e! Csak ebben az esetben írassuk ki!
5. A program kérdezzen két számot, s utána írja ki a köztük lévő relációt. Például, ha a két szám 3 és -6.12, akkor az eredmény: $3 > -6.12$.
6. Írj programot, ami egy életkor alapján eldönti, hogy gyerek (0-6 év), iskolás (7-18), dolgozó (19-60), illetve nyugdíjas-e az illető!
7. Fej vagy írás? A játék célja, hogy a játékos eltalálja, hogy a feldobott pénz fej vagy írás lesz. A játékos adjon tippet (fej, írás), majd a gép dobjon fel egy pénzérmét és írja ki, hogy a játékos nyert vagy veszített.
8. A gép dobjon dobókockával, majd két játékos tippelje meg a dobás eredményét. Az a játékos nyer, akinek a tippje közelebb van a kockadobás eredményéhez.
9. Adott egy pont, melynek bekérjük a koordinátáit. Határozzuk meg, melyik síknegyedben van!
10. Kérjünk be három számot, majd írassuk ki, hogy ezek lehetnek-e egy szerkeszthető háromszög oldalai!
11. Kérjünk be három számot, majd írjuk ki őket növekvő sorrendben!
12. Kérjük be egy másodfokú egyenlet együtthatóit, majd számoljuk ki az egyenlet gyökeit! Ha az együtthatók alapján az egyenlet nem megoldható írjunk ki hibaüzenetet!
13. Kérjünk be egy évszámot és döntsük el, hogy szökőév-e! Egy év akkor szökőév, ha az évszám maradék nélkül osztható 4-gyel, de nem osztható 100-zal, kivéve, ha az évszám osztható 400-zal.
14. Kérjük be egy hónap sorszámát, majd írjuk ki a nevét!
15. Kérjük be egy dolgozat pontszámát 1 és 100 között és írjuk ki az érdemjegyet! 0-40: elégtelen, 41-55: elégséges, 56-70: közepes, 71-85: jó, 86-100: jeles.

7 Ciklusok

Másik alapvető utasítástípus a programozási nyelvekben az ismétlés, más néven ciklus vagy iteráció. Akkor alkalmazzuk, amikor egy programrész többször kell ismételnünk. Az ismétlés történhet előre meghatározott lépésszámban vagy függhet egy feltételtől. Ide soroljuk azokat az utasításokat is, amik valamilyen, ún. iterálható adatszerkezet elemein lépkednek végig.

7.1 for

A `for (inicializálás; feltétel; növelés)` ciklus a JavaScript-ben is megtalálható:

```
for (let i = 1; i <= 5; i++) {  
  console.log(i);  
}
```

Valójában `let` nélkül is működik; ez esetben automatikusan úgy jön létre, mintha `var` lenne. A különbség az, hogy `let` nélkül az `i` definiált marad a ciklus után is, mégpedig 6 értékkel. Ez nem jó: a ciklus után nem szabad használni a ciklus változó értékét, és az a legjobb, ha a programozási nyelv nem is teszi ezt lehetővé. A `let` ezt a hibát is javította.

7.2 for ... in ...

A `for` ciklust a legritkább esetben használjuk számlálásra. Sokkal gyakoribb az, amikor egy struktúra elemein lépkedünk végig. Ez persze megvalósítható sima `for` ciklussal is, csak hogy annak több hátránya van:

- **Lassú.** Minden egyes lépésben megcímezzük a struktúra adott elemét, melynek ideje számottevő lehet.
- **Felesleges változó.** Létrehozunk egy változót, aminek csak az a szerepe, hogy végig iteráljunk az elemeken.
- **Nehezen olvasható.** Ha van egy bonyolultabb ciklus, és azon belül több utasítás, akkor első ránézésre sokszor nem egyértelmű, hogy itt egy struktúra elemein történő végiglépkedésről van szó.

Mint a legtöbb programozási nyelvben, a JavaScript-ben is bevezették a `for ... in ...` struktúrát. Ez szintaxisában eltérő az egyes nyelvekben, szemantikájában viszont ugyanazt jelenti.

```
let person = {name:"Pista", age:40, email:"pista@email.com"};  
  
for (key in person) {  
  console.log(key + " = " + person[key]);  
}
```

A példában létrehoztunk egy objektumot (amiről majd később lesz szó részletesen). Ebben kulcs-érték párok szerepelnek. A `for ... in ...` valójában a kulcsokon lépked végig. A kimeneten az alábbi eredmény jelenik meg:

```
name = Pista;  
age = 40;  
email = pista@email.com;
```


Nézzük meg mi történik, ha egy tömb elemein próbálunk végiglépkedni ezzel a módszerrel!

```
let cars = ["BMW", "Volkswagen", "Trabant"];

for (car in cars) {
  console.log(car);
}
```

Ha lefuttatjuk a kódot, akkor nem az autókat írja ki, hanem a következő számokat:

```
0
1
2
```

A JavaScript a tömböt is kulcs-érték párokként tárolja, melynek a kulcsa az index, az értéke pedig az adott indexű elem. Ezért a `for ... in ...` a kulcsokon lépked végig, az egyes iterációkban az `in`-től balra levő változó a kulcsokat, jelen esetben az indexeket fogja tartalmazni! A következő példa már az autók nevét írja ki:

```
let cars = ["BMW", "Volkswagen", "Trabant"];

for (car in cars) {
  console.log(cars[car]);
}
```

Az eredmény:

```
BMW
Volkswagen
Trabant
```

7.3 for ... of ...

A 2015-ben kiadott szabványban (ES6) bevezettek egy másik megoldást: `for ... of ...`. A szintaxisa ugyanaz, mint a `for ... in ...`-é, viszont ez már az értékeken lépked végig:

```
let cars = ["BMW", "Volkswagen", "Trabant"];

for (car of cars) {
  console.log(car);
}
```

Az eredmény:

```
BMW
Volkswagen
Trabant
```

Próbáljuk ki a korábbi példát itt is!

```
let person = {name:"Pista", age:40, email:"pista@email.com"};

for (key of person) {
  console.log(key + " = " + person[key]);
}
```

Az eredmény egy hibaüzenet, mely szerint az objektumunk nem iterálható típusú:

```
Uncaught TypeError: person is not iterable
```

Ennek oka, hogy a struktúráknak két fő csoportját különböztethetjük meg:

- **Enumerable** (megszámlálható): a `for ... in ...` ciklussal járható be, ami a *kulcsokon* lépked végig. Mivel az enumerable tágabb csoport, mint az iterable (ld. lejjebb), ezért azokon a struktúrákon, amelyek megszámlálhatóak, de nem bejárhatóak (pl. egy objektum elemei), a `for ... of ...` hibát jelez.
- **Iterable** (bejárható): kevesebb típust tartalmaz, mint az enumerable, ezért a `for ... in ...` a kulcsokon, a `for ... of ...` pedig az *értékeken* fog végiglépkedni.

Tehát azokon a struktúrákon, amelyek megszámlálhatóak, de nem bejárhatóak, a `for ... in ...`, azokon pedig, amelyek bejárhatóak, a `for ... of ...` szerkezetet kell használni.

7.4 while

Az elől tesztelőnek nevezett `while (feltétel) {ciklusmag}` ciklus igen gyakori. A ciklus mag addig hajtódik végre, amíg a ciklus elején a feltétel igaz. Ha a feltétek hamissá válik, a ciklusmag ismétlése befejeződik és a program futása a ciklus utáni utasításnál folytatódik. Ez azt is jelenti, hogy előfordulhat, hogy a ciklusmag egyszer sem fut le.

```
let i = 1;
while (i <= 5) {
  console.log(i);
  i++;
}
```

7.5 do ... while

A `do {ciklusmag} while (feltétel)` neve hátul tesztelő ciklus. Ritkábban alkalmazzuk, mint az elől tesztelőt. Nem is minden programozási nyelvben szerepel, de a JavaScript-ben megtalálható. Ebben az esetben a ciklusmag legalább egyszer lefut, majd a végén történik a feltétel ellenőrzés, és amíg a feltétel igaz, ismétlődik a ciklusmag futása.

```
let i = 1;
do {
  console.log(i);
  i++;
} while (i <= 5);
```

7.6 Ugró utasítások

A JavaScript-ben nincs `goto` utasítás, `break` és `continue` viszont van.

A `break` kilép a ciklusból. Tipikusan feltételen belül hívjuk meg:

```
let i = 1;
while (true) {
  if (i > 5) {
    break;
  }
  console.log(i);
  i++;
}
```

A `continue` a ciklusmag elejére ugrik. Az alábbi példa a kiírásban kihagyja a páratlan számokat:

```
let i = 0;
while (i < 10) {
  i++;
  if (i % 2 == 1) {
    continue;
  }
  console.log(i);
}
```

7.7 Feladatok

1. Írassuk ki 1-től 20-ig a számokat, a négyzetüket és a négyzetgyöküket táblázatos formában!
2. Tetszőleges betűvel tetszőleges (1-10) sort töltsünk fel a képernyőn!
3. Írassuk ki 99-től csökkenő sorrendben az összes pozitív, 3-mal osztható egész számot!
4. Írassuk ki 101-től 50-ig csökkenő sorrendben az ötten osztható számok kétszeresét!
5. Határozzuk meg az első N természetes szám összegét!
6. Írjuk ki az első N négyzetszám átlagát!
7. Számítsa ki a gép 10 tetszőleges szám összegét, szorzatát és átlagát!
8. Készítsünk N-es szorzótáblát ($1 \times N$, $2 \times N$,...)!
9. Készítsen a gép számtani sorozatot (adott az első elem, a differencia és elemszám)!
10. Szimuláljunk egy N-szeres kockadobást: dobjunk fel N-szer egy kockát a gép segítségével. Írjuk ki az egyes dobások eredményét, majd a sorozat végén a dobások átlagát is!
11. Szimuláljunk kockadobást: dobjunk fel addig a kockát, amíg hatost nem dobunk. Írjuk ki az egyes dobások eredményét, majd a sorozat végén a dobások átlagát is!
12. Szimuláljunk kockadobást: dobjunk fel addig a kockát, amíg 3 db hatost nem dobunk. Írjuk ki az egyes dobások eredményét, majd a sorozat végén a dobások átlagát is!
13. Szimuláljunk kockadobást: dobjunk fel addig a kockát, amíg a dobások összege kisebb, mint K. Írjuk ki az egyes dobások eredményét, majd a sorozat végén a dobások átlagát is!
14. Kérjen be a program számokat mindaddig, amíg 0-át nem írunk be! Ezután írja ki, hogy páros szám volt a beírt számok között!
15. Egy tetszőleges számot addig harmadoljunk, amíg 0,01-nél kisebb értéket nem kapunk! Írjuk ki a harmadolások számát!
16. Írj programot, ami a barchoba játékot valósítja meg! A gép „gondoljon” egy számra 1 és 100 között. Miután tippeltünk adjon választ, hogy kisebbre vagy nagyobbra gondolt. Ha eltaláltuk a gondolt számot, jelezze a találatot!
17. Írj programot, ami egy bekért számról eldönti, hogy prímszám-e!
18. Kérj be egy számot és írasd ki a prímtényező felbontását!
19. Írasd ki két szám legnagyobb közös osztóját! A megoldáshoz használhatod az Euklideszi algoritmust.

8 Stringek kezelése

Mi az a string? A legtöbb programnyelvben már a tanulás elején felmerül ez a téma. A JavaScript-ben is jelen vannak a stringek, de itt kisebb hangsúlyt kapnak, mert nem kell a változó deklarálásakor megadni a változó típusát. Szóval ugyanúgy létrehozunk string-eket, csak sokszor nem nevezzük a nevükön őket.

A programozás területén a karakterek sorozatát string-eknek (vagy magyarul sztringeknek, karakterláncoknak és nagyon ritkán karakterfüzérnek is) nevezzük. Fogalmazhatunk úgy is, hogy a karakterek tárolására alkalmas adattípust nevezzük string-eknek.

8.1 String létrehozása JavaScript-ben

String létrehozása JavaScriptben:

```
let s = "karakterlánc";
```

Létrehozunk egy változót vagy konstanst, majd megadjuk a változó nevét és egy egyenlőségjel után idézőjelek között az értékét.

8.2 Escape karakterek

A feloldójel (angolul escape character) olyan karakter, amely egy karakterláncban azt jelöli, hogy a következő karaktert másképpen kell értelmezni, mint általános esetben.

A JavaScript nyelvben a konzolra íráskor az alábbi lehetőségeket alkalmazhatjuk.

Escape kód	Hatása
\'	Aposztróf
\"	Idézőjel
\\	Backslash karakter
\b	Backspace (vissza törlés)
\n	Új sor
\r	Kocsi vissza
\t	Vízszintes tabulátor

8.3 A String metódusai és tulajdonságai

Nagyon leegyszerűsítve a metódusok a teljes programkód egy apróbb részfeladatát végzik el. A metódus (vagy a függvény) általában valamilyen komplexebb műveletet végez el, például sorba rendezi egy objektum elemeit, míg a tulajdonság csak egy információt ad vissza az objektumról, például, hogy hány elemből áll. (A metódusokról és tulajdonságokról bővebben olvashat az objektumok fejezetben.)

A `length` tulajdonság arra alkalmas, hogy megállapítsa egy string hosszát, azaz visszaadja a szövegben lévő karakterek számát.

Mivel ez egy tulajdonság, így a `length` kulcsszó után nem kell zárójel. Fontos tudni, hogy ez a tulajdonság csak lekéri a string karaktereinek a számát, de sehol sem tárolja, és sehova sem írja ki őket, így azzal még foglalkoznunk kell.

Ezek a tulajdonságok bármilyen szövegen alkalmazhatók. Ez lehet egy változóban tárolt szöveg és egy idézőjelek közé írt szöveg is.

```
let s = "JavaScript";
console.log(s.length);
console.log("JavaScript".length);
```

A string betűi indexeltek ezért egy ciklussal kiírhatjuk őket.

```
let s = "JavaScript";
for(let i = 0; i < s.length; i++) {
  console.log(s[i]);
}
```

A stringek iterálható objektumok, ezért a `for ... of` ciklus is alkalmazható a betűk kiírására.

```
let s = "JavaScript";
for(betu of s) {
  console.log(betu);
}
```

A `concat()` metódus segítségével számokat, szövegeket, karaktereket tudunk összefűzni. Két paramétere van. Az első az összefűzendő jelek közötti elválasztó karakterek, a második az összefűzendő szöveg. Tömbök összefűzésére is alkalmazható.

```
let s1 = "Tanulj";
let s2 = "JavaScript-et!";
let s = s1.concat(" ", s2);
console.log(s);
```

A `charAt()` metódus visszaadja a paraméterként megadott index helyén lévő karaktert. A `charCodeAt()` metódus az indexelt karakter unicode-ját adja vissza.

```
let s = "JavaScript";
console.log(s.charAt(4));
console.log(s.charCodeAt(4));
```

Az `endsWith()` metódus igaz (true) értékkel tér vissza, ha a string a metódus paraméterében megadott stringre végződik. A `startsWith()` a string elején keresi az egyezést.

```
let s = "Helló világ!";
console.log(s.endsWith("világ!"));
console.log(s.startsWith("világ!"));
```

A `fromCharCode()` metódus a paraméterként kapott karakterkódokat karakterekké alakítja. Az átalakítás a unicode kódtábla alapján történik.

```
console.log(String.fromCharCode(65, 108, 109, 97));
```

Az `includes()` metódus igaz értéket ad vissza, ha a paraméterként kapott string megtalálható a vizsgált stringben. Opcionálisan megadható a keresés kezdőindexe is.

```
let s = "Az alma nem esik messze a fájától.";
console.log(s.includes("alma"));
console.log(s.includes("alma", 7));
```

Az `indexOf()` metódus a paraméterként megadott string első, míg a `lastIndexOf()` metódus az utolsó előfordulását adja vissza. Opcionálisan megadható a keresés kezdőindexe. Amennyiben a keresés sikertelen, a metódus -1 értékkel tér vissza. A `lastIndexOf()` metódus a string végétől kezdi a keresést.

```
let s = "Az alma nem esik messze a fájától.";
console.log(s.indexOf("alma"));
console.log(s.indexOf("a", 7));
console.log(s.lastIndexOf("a"));
console.log(s.lastIndexOf("a", 7));
```

A következő két metódus az első paraméterben megadott hosszra bővíti a stringet, a második paraméterben megadott string ismétlésével. A `padEnd()` a string végéhez, a `padStart()` a string elejéhez fűzi a karaktereket, míg a megadott hosszt el nem érik.

```
let s = "10";
console.log(s.padEnd(5, "0"));
console.log(s.padStart(5, "0"));
```

A `repeat()` metódus a stringet többszörözi, a paraméterben megadott érték szerint.

```
let s = "1a";
console.log(s.repeat(5));
```

A `replace()` metódussal egy, a string-ben szereplő szót cserélhetünk le egy általunk választott szóra. A csere folyamán csak a keresett szó első előfordulása kerül lecserélésre. A `replaceAll()` metódus az összes előfordulást cseréli. (lásd még a reguláris kifejezéseknél)

```
let szoveg="A kedvenc programozási nyelvem a HTML és a HTML.";
console.log(szoveg);
uj_szoveg = szoveg.replace("HTML", "JS");
console.log(uj_szoveg);
uj_szoveg = szoveg.replaceAll("HTML", "JS");
console.log(uj_szoveg);
```

A következő vizsgált metódusok az `s` string egy rész stringjét adják vissza. A `slice()` metódus az első paraméterként adott indexű karaktertől a második paraméterben megadott indexű karakterig adja vissza a string egy részét. Ha egy paramétert adunk meg, akkor ettől a karaktertől a string végéig kapjuk a rész stringet. Amennyiben negatív értéket adunk meg, úgy a string végétől kezdjük a számolást. A `substring()` metódus annyiban különbözik, hogy a negatív értéket 0-ként kezeli. A `substr()` metódus első paramétere az előzőekhez hasonlóan a rész string kezdőindexe, a második paraméter a visszaadandó karakterek száma.

```
let s = "JavaScript";
console.log(s.slice(3, 6));      //aSc
console.log(s.slice(4));        //Script
console.log(s.slice(-6));       //Script
console.log(s.slice(-7, -4));   //aSc
console.log(s.substring(2, 7)); //vaScr
console.log(s.substr(3, 3));    //aSc
```

Mint ahogy a legtöbb programnyelvben itt is nullától indul a számozás, azaz a „JavaScript” szóban az első karakter, a „J” a 0. indexű helyen áll.

A JavaScript `split()` metódusa egy megadott karakter mentén darabolja fel a szöveget. Azaz, ha van egy szövegünk, amiben szóközzel vannak elválasztva a szövegek, akkor feldarabolhatjuk a szöveget a szóközők mentén. Ennek az lesz az eredménye, hogy létrejön egy

tömb, amelyben szerepel az összes olyan szó, amelyet szóközök választottak el. (lásd még a reguláris kifejezéseknél)

```
let v = "A JavaScript egy szkriptnyelv.";
let szavak = v.split(" ");
for(szo of szavak) {
    console.log(szo);
}
```

Ahogy a nevük is sugallja, a `toUpperCase()` nagybetűssé, míg a `toLowerCase()` kisbetűssé teszi a szövegünket.

```
console.log("JavaScript".toUpperCase());
console.log("JavaScript".toLowerCase());
```

A `trim()` metódus segítségével a „white space” karaktereket tüntethetjük el a szöveg elejéről és végéről. A „white space”-t magyarul fehér szóköznek vagy üres helynek is szoktuk nevezni. A programozásban azokat a karaktereket értjük ezalatt, amelyek nem láthatóak a szövegben. A `trimEnd()` csak a karakterlánc végéről, a `trimStart()` csak a karakterlánc elejéről távolítja el a „white space” karaktereket.

```
console.log("    Java    Script    ".trim());
```

8.4 Feladatok

1. Írj programot, ami bekéri a felhasználó bevét, majd keresztnévén szólítva köszönti!
2. Írj programot, mely megszámolja, hogy az inputként érkező mondatban hány darab „a” betű van!
3. Olvass be egy szöveget, és írd ki a betűit fordított sorrendben!
4. Olvass be egy mondatot és egy szót! Írasd ki, hogy a szó szerepel-e a mondatban!
5. A beolvasott mondatról dönts el, hogy az visszafelé is ugyanazt jelenti-e! (Az „Indul a görög aludni”, vagy a „Géza kék az ég” visszafelé olvasva is ugyanazt jelenti.) Ügyelj a mondatvégi írásjelekre, mivel azok a mondat elején nem szerepelnek.
6. Az inputként beolvasott szövegben cseréld ki az összes szóközt a # karakterre, majd az így kapott szöveget írd a képernyőre!
7. Olvass be egy mondatot, és írd ki a szavait egymás alá.
8. Olvass be egy mondatot, és írd ki a szavait egymás alá úgy, hogy a szavak első betűje nagybetűs legyen!
9. Kérj be egy nevet és írasd ki a monogramját! A program akkor is helyesen működjön, ha a név kettős mássalhangzóval kezdődik.
10. Olvass be egy szöveget és két karaktert! A szövegben az első karakter összes előfordulását cseréld a második karakterre!
11. Olvass be egy karakterláncot „123-23-12-4-5-65” formátumban! Írd ki a benne szereplő számok összegét!
12. Kérd be N értékét és írasd ki egy N db véletlen karakterből álló karakterláncot!
13. Kérj be szót és írasd ki a betűi UNICODE-ját egymás alá!
14. Írj programot, ami egy beírt szövegben az összes kisbetűt nagybetűre, a nagybetűket pedig kisbetűre cseréli!
15. Írj programot, ami bekér két karakterláncot és kiírja, hogy az egyik tartalmazza-e a másikat!

9 Tömbök

9.1 Mik azok a tömbök?

A tömb egy speciális változó, amely egyidejűleg több elemet képes tárolni. A tömb (angolul array) olyan adatszerkezet, amelyet nevesített elemek csoportja alkot, melyekre sorszámukkal (indexükkel) lehet hivatkozni.

9.2 Mire való a tömb?

Olyan esetekben használjuk, amikor több elem szerepel egy listán, általában az elemek között valamilyen összefüggés van. Például van egy listánk, amely különböző webes programnyelveket tartalmaz. Itt a kapcsolódási pont az, hogy minden elemünk egy webes programnyelv.

A fent említett elemeket (programnyelveket) tárolhatnánk az alábbi módon is:

```
let nyelv1 = "HTML";  
let nyelv2 = "CSS";  
let nyelv3 = "JS";
```

A fenti példával sincs semmilyen gond, működik, rendesen lefut, de be kell látnunk, hogy egy kicsit sok időbe telik, mire egyesével létrehozunk annyi változót, ahány elemünk van. És akkor még nem is beszéltünk arról, hogy mi van akkor, ha az összes elemet meg szeretnénk vizsgálni. Nehéz lenne rá olyan módszert találni, ami gyors is és egyszerű.

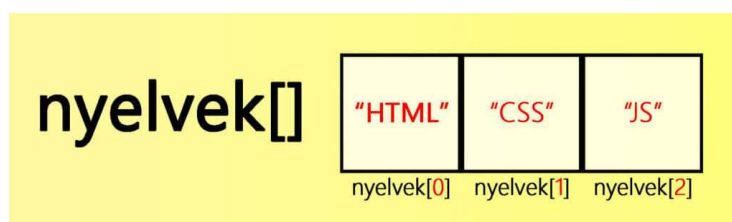
Ezekre a problémákra adnak megoldást a tömbök. Ha egy tömböt hozunk létre, akkor nincs szükség arra, hogy minden egyes programnyelvet külön változóban tároljunk, szimplán létrehozunk egy tömböt és abba pakoljuk bele az összes értéket, amit alapesetben külön-külön változóban tárolnánk.

A fenti példa így néz ki tömbbel megoldva:

```
let nyelvek = ["HTML", "CSS", "JS"];
```

9.3 Mi az az index?

A tömb elemeire a sorszámukkal lehet hivatkozni. Először tekintsük meg az alábbi ábrát.



Van egy tömbünk, annak vannak elemei, amelyekre sorszámokkal hivatkozunk, ezeket a sorszámokat nevezzük indexnek. Láthatjuk, hogy a tömbünk első eleme a „HTML”. Az informatikában a számozás nem egytől, hanem nullától kezdődik! Így az első sorszámunk nem 1, hanem 0 lesz. Azaz a „HTML” elemünket úgy tudjuk elérni, ha lekérjük a `„nyelvek[0]”` tömbünk nulladik indexű elemét.

Ha a „JS” után szeretnénk berakni egy újabb elemet, mondjuk a „PHP”-t, akkor az utolsó elem után következő elemre kell hivatkoznunk, jelen esetben a 3. indexre.

9.4 Tömb létrehozása

A `let`, `const` vagy `var` kulcsszó után megadjuk a tömb nevét, majd az egyenlőségjel után két szögletes zárójel között felsoroljuk a tömbünk elemeit. Ezt hívjuk az úgynevezett ömlesztett tömb deklarációnak.

```
let nyelvek = ["HTML", "CSS", "JS"];
```

Egy tömb létrehozása során az elemek felsorolásánál nem számítanak a szóközök, ezért így is létrehozhatjuk a tömbünket:

```
let nyelvek = [  
  "HTML",  
  "CSS",  
  "JS"  
];
```

Természetesen egy tömb nem csak három elemet tartalmazhat, nyugodtan írhatunk negyedik, ötödik vagy százhuszonhatodik elemet is.

Úgy is létrehozhatunk egy tömböt, hogy nem adunk neki kezdő elemeket, csak utólag adjuk hozzá azokat. Ez így néz ki:

```
let nyelvek = [];  
  
nyelvek [0] = "HTML";  
nyelvek [1] = "CSS";  
nyelvek [2] = "JS";
```

Legutolsó megoldásként létrehozhatunk a „`new Array()`” kulcsszóval is tömböt, de ez is ugyanazt csinálja, mint a fenti megoldások. Az egyszerűség, olvashatóság és a futási sebesség érdekében nem szokták ezt a megoldás használni.

```
let nyelvek = new Array("HTML", "CSS", "JS");
```

9.5 Tömb elemének elérése

Fentebb volt már szó az indexelésről, ami elengedhetetlen egy elem eléréséhez, hiszen ha nem tudjuk az index számát, akkor nem is tudunk hozzáférni.

Egy elemet egyszerűen úgy érhetünk el, ha megadjuk, hogy a tömb hányadik indexére vagyunk kíváncsiak, ezt így tehetjük meg:

```
let nyelvek = ["HTML", "CSS", "JS"];  
  
console.log(nyelvek[0]);  
console.log(nyelvek[1]);  
console.log(nyelvek[2]);
```

9.6 Tömb összes elemének elérése

Lehetőségünk van arra is, hogy a tömbnek ne csak egyetlenegy elemét kérjük le, hanem egyszerre az összeset.

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log(nyelvek);
```

9.7 Tömb elemének módosítása

Ahhoz, hogy módosítsunk egy tömbelemet, tisztában kell lennünk az indexével, vagy ha nem vagyunk tisztában vele, akkor ki kell derítenünk.

Ha megvan az index, akkor az alábbi utasítással módosíthatjuk a tömb elemét.

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log("Módosítás előtt: " + nyelvek[1]) ;
nyelvek[1] = "PHP";
console.log("Módosítás után: " + nyelvek[1]);
```

Példánkban a nyelvek tömb első indexű elemét cseréljük le a „PHP” szóra.

9.8 Új elem hozzáadása a tömbhöz

Előfordulhat velünk olyan eset is – valójában szinte ez a leggyakoribb –, hogy egy létező tömbhöz újabb elemeket szeretnénk hozzáadni. Ennek megvalósítására több módszer is a rendelkezésünkre áll.

A legegyszerűbb módszer a `push()` metódus használatával történik. Ilyenkor nem szükséges megadnunk, hogy hova szeretnénk a tömbben elhelyezni az új elemet, mert az automatikusan elhelyezi a legutolsó elem után. Egyszerűen megadjuk, hogy mit szeretnénk elhelyezni a tömbben, a többit pedig a JavaScript-re és a böngészőnkre bízhatjuk.

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log(nyelvek);
nyelvek.push("PHP"); //itt adunk hozzá egy újabb elemet a tömbhöz
console.log(nyelvek);
```

A `push()` esetében nem mondjuk meg, hogy hova tegye az értéket, csak azt, hogy van egy új érték, amit hozzá kell adni a tömbhöz. Az új érték minden esetben a tömb végére kerül.

A következő módszernél megmondjuk, hogy pontosan hova tegye a tömbhöz hozzáadni kívánt értéket. Mivel új elemet szeretnénk hozzáadni a tömbhöz, meg kell állapítanunk, hogy mi a tömb utolsó elemének indexe, hogy ennél eggyel nagyobb indexű helyre tudjunk értéket betenni. A `length` a tömb hosszát adja vissza (később részletezzük a működését), ami alkalmas az index meghatározására.

A „`nyelvek.length`” gondoskodik arról, hogy az új elemünk a tömb utolsó eleme után legyen elhelyezve. Természetesen erre is láthatok egy példát:

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log(nyelvek);
nyelvek[nyelvek.length] = "PHP"; //új elemet adunk a tömbhöz
console.log(nyelvek);
```

9.9 Tömb hosszának megállapítása

Fentebb szó volt a `length` tulajdonságról. Ez a tulajdonság adja vissza egy tömb hosszát, vagyis a tömb elemeinek számát.

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log(nyelvek.length);
```

9.10 Tömbelem törlése

Ha az utolsó elemet szeretnénk törölni, akkor érdemes a `pop()` metódust használnunk, ami körülbelül ugyanúgy működik, mint a `push()`, csak ez nem az utolsó helyre tesz be, hanem töröl onnan.

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log(nyelvek);
nyelvek.pop();
console.log(nyelvek);
```

Most pedig jöjjön a kevésbé elegáns törlési módszer, sokan nem is ajánlják használni, mert „lyukassá” válik a tömbünk, azaz több helyen „undefined” érték lesz, de sajnos vannak olyan helyzetek, amikor kénytelenek vagyunk ezt használni.

Megadjuk a „`delete`” kulcsszót, majd a tömbnevet, melyből törölni szeretnénk és azt, hogy melyik indexére vonatkozik a törlés.

```
let nyelvek = ["HTML", "CSS", "JS"];
console.log(nyelvek);
delete nyelvek[1];
console.log(nyelvek);
```

A hiányzó elem helyén az „empty” (üres) szó jelenik meg a konzolon.

9.11 Asszociatív tömb

A tömbök indexelésére pozitív egészek helyett használhatunk Stringeket is. Ekkor használható a `.` operátor az egyes tömbelemek eléréséhez.

```
let myArray = new Array();
myArray["a"] = 100;
console.log(myArray["a"]);
console.log(myArray.a);
```

Innen már látható, hogy a háttérben valójában egy `"a"` nevű tulajdonságot hoztunk létre a `myArray` objektumon a 100 értékkel, mely kétféle szintaxis szerint érhető el. Ebből következik, hogy ezt bármilyen objektummal megtehetjük, így bármilyen objektum használható asszociatív tömbként. Mégis a tömböket érdemes erre a célra használni, egyrészt az átláthatóság miatt, másrészt, a tömb objektum plusz szolgáltatásai (pl. a rendezésre: `sort()` függvény) miatt.

9.12 Többdimenziós tömbök

Az eddigiekben egydimenziós tömbökkel, más néven vektorokkal foglalkoztunk. Ezeknél minden tárolt adatra egy indexszel hivatkozhatunk. A kettő vagy több dimenziós tömböket mátrixoknak is nevezzük. Ezeknél az adatokra több indexszel hivatkozunk, attól függően, hány

dimenziós a tömb. Két, esetleg három dimenziós tömböket még szoktunk alkalmazni. Ennél nagyobb tömbökre csak nagyon speciális esetben lehet szükségünk.

Az egydimenziós tömböket vektornak, a több dimenziós tömböket mátrixnak is szokták nevezni.

A JavaScript a többdimenziós tömbök kezelését egydimenziós tömbök egymásba ágyazásával oldja meg.

```
let tomb2D = [
  [1, 3, 4, 6],
  [5, 7, 2, 3],
  [8, 9, 3, 1]
];
```

A tömb egy elemét a következő módszerrel írathatjuk ki:

```
console.log(tomb2D[1][1]); //7
```

Az indexek itt is 0-val kezdődnek, ezért az 1-es indexű sor, 1-es indexű eleme a 7 lesz.

A mátrix elemein két ciklus egymásbaágyazásával tudunk végiglépkedni. A külső ciklus a tömb sorait veszi sorra, míg a belső az aktuális sor elemein lépked végig. Mivel a `console.log()` minden értéket egymás alá ír ki, a kívánt formátum eléréséhez egy stringbe fűzzük a mátrix elemeit.

```
let matrix = "";
for (let sor of tomb2D) {
  for (let oszlop of sor) {
    matrix += oszlop + " ";
  }
  matrix += "\n";
}
console.log(matrix);
```

9.13 Tömbök metódusai

Az `at()` metódus a paraméterként megadott indexű elemet adja vissza a tömbből. Alapértelmezett értéke a 0. Ha a paraméter nem negatív, akkor ugyanazt az eredményt adja, mint a „`[]`”. Negatív index esetén a tömb végéről kezdi az indexelést visszafelé. Ebben az esetben a -1 index a tömb utolsó elemére mutat.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo"];
console.log(autok.at(2)); //Audi
console.log(autok.at(-3)); //BMW
```

A `concat()` metódus egyetlen tömbbé fűzi össze a paramétereket, melyek lehetnek tömbök, de egyszerű értékek is. A metódus az eredeti tömbök értékét nem változtatja meg.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo"];
let buszok = ["Setra", "Neoplan", "MAN"];
let traktor = "Johndeer";
let jarmuvek = autok.concat(buszok, traktor);
console.log(jarmuvek);
```

A `fill()` metódus a paraméterként megadott értékkel tölti fel a tömb meghatározott elemeit. Az első paraméter az az érték, amivel a tömb elemeit fel akarjuk tölteni. Ha nem adunk meg több paramétert, a teljes tömböt feltölti a megadott értékkel. A második és harmadik

paraméterrel megadhatjuk, hogy mettől-meddig töltsük fel az új értékkel a tömb elemeit. Ezen két paraméter megadása nem kötelező. A módszer az eredeti tömb értékeit írja felül.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo", "Renault"];
autok.fill("Trabant");
console.log(autok);
autok.fill("Lada", 2, 4);
console.log(autok);
```

A `from()` módszer bármilyen iterálható objektumot tömbbé alakít.

```
let szoveg = "ABCDEFGH";
let betuk = Array.from(szoveg);
console.log(betuk);
```

A `join()` módszer a tömb elemeinek felsorolásából álló Stringet ad vissza, melyben az elemeket a paraméterül átadott szöveg választja el.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo", "Renault"];
console.log(autok.join(" - "));
```

A `reverse()` módszer megfordítja a tömb elemeinek sorrendjét.

```
let szamok = [5, 8, 3, 12, 31];
szamok.reverse();
console.log(szamok);
```

Az `indexOf()` módszer megkeresi a paraméterként kapott érték első előfordulását a tömbben. Opcionálisan megadhatjuk a keresés kezdőindexét is. Ha a keresett elem nem található, akkor -1 értékkel tér vissza.

A `lastIndexOf()` módszer az utolsó előfordulás indexét adja vissza.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo", "Renault", "BMW"];
console.log(autok.indexOf("BMW"));
console.log(autok.indexOf("Mercedes", 2));
console.log(autok.lastIndexOf("BMW"));
console.log(autok.lastIndexOf("BMW", 2));
```

Az `includes()` módszer egy paraméterként kapott értéket keres a tömbben, vagy annak egy részében. Ha a keresés eredménye sikeres „true”, ha nem, akkor „false” értékkel tér vissza. Második paraméterként megadhatunk egy tömbindexet. Ebben az esetben ettől az indexű elemről kezdi a keresést.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo", "Renault"];
console.log(autok.includes("BMW"));
console.log(autok.includes("BMW", 2));
```

A `slice()` módszer az egész tömbről, vagy a paraméterek által megadott indexhatárok közötti részből másolatot készít. A tömbök referencia szerint adódnak át (lásd: Függvények). Az érték szerinti átadás szimulálására is használhatjuk a `slice()` függvényt, ami egy másolatot készít az egész tömbből.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo", "Renault"];
console.log(autok.slice(1, 3));
console.log(autok.slice(-3, -1));
```

A `sort()` metódus rendezi a tömb elemeit. Ha nem adunk meg paramétert a rendezés alfanumerikus lesz. Számok rendezésénél ez hibás eredményt adhat. Pl.: a 12 a 3 előtt fog szerepelni, mert az „1” a „3” előtt szerepel a karaktertáblában.

A számok rendezése is megoldható, ha a megfelelő függvényt alkalmazzuk a metódus paramétereként. A függvény a számok különbsége alapján dönti el a sorrendet, attól függően, hogy a visszaadott érték „+”, „-” vagy „0”.

```
let autok = ["Mercedes", "BMW", "Audi", "Volvo", "Renault"];
autok.sort();
console.log(autok);
let szamok = [5, 8, 3, 12, 31];
szamok.sort();
console.log(szamok);
szamok.sort(function(a, b){return a - b});
console.log(szamok);
```

A verem egy olyan adatszerkezet, amelyből mindig csak az utolsónak behelyezett elemet tudjuk kivenni. A szakirodalom ezt a megoldást LIFO (Last In, First Out) szerkezetnek nevezi. A verem szerkezet két alapvető művelettel rendelkezik. Ezek a `push()` és a `pop()`. A `push()` művelet hozzáad egy elemet a verem tartalmához, míg a `pop()` művelet kiveszi a veremből a legfelső, legutoljára behelyezett elemet. A JavaScript nyelvben a verem adatszerkezetet is tömb használatával valósíthatjuk meg. A JavaScriptben a tömbök rendelkeznek `push()`, és `pop()` műveletekkel, melyek a tömb végére tesznek be egy elemet, illetve vesznek el onnan.

```
let szamok = [5, 8, 3, 12, 31];
szamok.push(2);
szamok.push(45, 10);
console.log(szamok);
console.log(szamok.pop());
console.log(szamok);
```

A `shift()` metódus kilépteti az első elemet a tömbből, míg az `unshift()` belépteti a tömb elejére a tetszőleges számú paraméterét.

```
let szamok = [5, 8, 3, 12, 31];
szamok.unshift(2);
szamok.unshift(45, 10);
console.log(szamok);
console.log(szamok.shift());
console.log(szamok);
```

Az `every()` metódus ellenőrzi, hogy a paraméterként kapott, logikai értéket visszaadó függvény a tömb összes elemére igaz-e. Ha egy elemre hamis a függvény értéke, úgy a kifejezés értéke is hamis lesz.

```
let szamok = [5, 8, 3, 12, 31];
console.log(szamok.every(nagyobb)); //false

function nagyobb(szam) {
    return szam > 6;
}
```

A `some()` metódus a paraméterül kapott függvényt addig alkalmazza a tömb elemeire, míg a függvény igazat nem ad. Ekkor a visszatérési érték igaz, egyébként hamis.

```
let szamok = [5, 8, 3, 12, 31];
console.log(szamok.some(nagyobb));

function nagyobb(szam) {
    return szam > 12;
}
```

A `find()` metódus visszaadja az első olyan értéket a tömb elemei közül, amelyre a paraméterül megadott logikai értéket visszaadó függvény igaz értékkel tér vissza. Amennyiben nem talál ilyen értéket, „undefined” értékkel tér vissza. A `findIndex()` metódus annyiban különbözik, hogy nem az elemet, hanem annak indexét adja vissza. Ha a tömbben nincs a függvény feltételének megfelelő érték, akkor -1 értékkel tér vissza.

```
let szamok = [5, 8, 3, 12, 31];
console.log(szamok.find(nagyobb));
console.log(szamok.findIndex(nagyobb));

function nagyobb(szam) {
    return szam > 6;
}
```

A `filter()` metódus a paraméterül kap egy logikai értékkel visszatérő függvényt. Visszaad egy tömböt, ami azokat az elemeket tartalmazza, amire a függvény igaz értéket ad.

```
let szamok = [5, 8, 3, 12, 31];
console.log(szamok.filter(nagyobb));

function nagyobb(szam) {
    return szam > 6;
}
```

A `forEach()` metódus a paraméterül kapott függvényt a tömb minden elemére végrehajtja.

```
let kimenet = "";
let szamok = [5, 8, 3, 12, 31];
szamok.forEach(duplaz);
console.log(kimenet);

function duplaz(ertek, index) {
    kimenet += index + ": " + ertek * 2 + "\n";
}
```

A `map()` metódus a paraméterül kapott transzformáló függvényt alkalmazza a tömb minden elemére, és a transzformált elemek tömbjével tér vissza.

```
let szamok = [5, 8, 3, 12, 31];
let gyok = szamok.map(Math.sqrt);
console.log(gyok);
console.log(szamok.map(duplaz));

function duplaz(szam) {
    return szam * 2;
}
```

9.14 Feladatok

1. Szimulálj 100 dobást hat oldalú dobókockával! Készíts statisztikát a dobásokról! Végül írd ki, hogy hány átlag feletti dobás volt!
2. Tölts fel egy 100 elemű tömböt -50 és 50 közötti véletlenszámokkal!
 - a. Írased ki a tömb legnagyobb értékű elemét és annak indexét!
 - b. Írased ki a tömb elemeinek összegét!
 - c. Írased ki a páros és a páratlan elemek darabszámát!
 - d. Van-e a tömbben 7-tel osztható elem?
 - e. Van-e a tömbben olyan elem, amelynek mindkét szomszédja negatív?
 - f. Van-e a tömbben olyan elem, amely nagyobb, mint két szomszédjának összege?
 - g. Írased ki a tömb utolsó 3-mal osztható, de 5-tel nem osztható elemének indexét!
 - h. Van-e a tömbben 3 egyforma szám?
 - i. Van-e a tömbben egymás melletti azonos érték?
 - j. Írd ki azon számok indexeit, amelyek 10 többszörösei!
 - k. Hány átlag alatti szám van a tömbben?
3. Írj programot, ami bekér egy egész számot és egy számrendszer alapját (1-32)! A bekért számot alakítsd a megadott számrendszerűre, majd írased ki!
4. Egy 3x3-as mátrixot tölts fel 0 és 9 közötti véletlenszámokkal, majd írased ki!
 - a. Számold ki a számok összegét soronként!
 - b. Számold ki a számok összegét oszloponként!
 - c. Számold ki az átlók elemeinek összegét!
5. Kérjük be egy hét hőmérsékleti adatait, majd írássuk ki a heti legnagyobb hőingadozást!
6. A TAJ szám egy kilenc számjegyből álló szám, amelyben az első nyolc számjegy egy folyamatosan kiadott egyszerű sorszám, amely mindig az előző, utoljára kiadott sorszámból egy hozzáadásával keletkezik. A kilencedik számjegy ellenőrző ún. CDV kód, melynek képzési algoritmusa az alábbi:

A TAJ szám első nyolc számjegyéből a páratlan helyen állókat hárommal, a páros helyen állókat héttel szorozzuk, és a szorzatokat összeadjuk. Az összeget tízzel elosztva a maradékot tekintjük a kilencedik, azaz CDV kódnak.

Írj programot, amely bekér egy TAJ számot és ellenőrzi azt a fenti algoritmussal!
7. Tölts fel egy 1000 elemű tömböt 0 és 9 közötti véletlenszámokkal! Írd ki, hogy mettől meddig tart a leghosszabb azonos számokat tartalmazó szakasz!

10 Halmazok

A halmaz egyedi értékek tárolására alkalmas adatszerkezet, melyben minden érték csak egyszer fordulhat elő. A halmaz bármilyen típusú értéket tárolhat.

A halmaz létrehozása a `new Set()` konstruktorral történik.

```
let betuk = new Set(['a', 'b', 'c', 'd']);
```

A halmaz elemei egy `for ... of` ciklussal kiíratathatók:

```
for (let elem of betuk) {  
  console.log(elem);  
}
```

10.1 A halmazok tulajdonságai, metódusai

Új értéket az `add()` metódussal adhatunk a halmazhoz, melynek paramétere lehet konstans vagy változó is.

```
betuk.add('e');  
let a = 'f';  
betuk.add(a);
```

Ha ugyanazt az értéket többször adjuk a halmazhoz, az csak egyszer kerül tárolásra.

```
betuk.add('e');  
betuk.add('e');
```

A `size` tulajdonsággal lekérhetjük a halmaz elemeinek számát.

```
console.log(betuk.size);
```

A `forEach()` metódus a tömbhöz hasonlóan a halmaz elemeire is meghív egy függvényt. Ezt felhasználhatjuk arra is, hogy kiíratjuk az elemeket.

```
betuk.forEach(function(ertek) {  
  console.log(ertek);  
});
```

A `has()` metódus igaz értékkel tér vissza, ha a paramétereként megadott elem része a halmaznak.

```
if (betuk.has('c')) {  
  console.log('Ez az elem a halmazban van!');  
}
```

A `delete()` metódus egy, a paramétereként megadott elemet töröl a halmazból. A `clear()` metódus a halmaz összes elemét törli.

A `values()` metódus egy iterálható objektummal tér vissza, amely tartalmazza a halmaz minden elemét a beillesztés sorrendjében.

```
let iterator = betuk.values();  
console.log(iterator.next().value);  
console.log(iterator.next().value);
```

A halmazok elemeihez nem tartoznak kulcsok, így a `keys()` metódus is a tárolt elemekkel tér vissza. Az `entries()` metódus kulcs-érték párokkal tér vissza, de a halmazokban a kulcs hiánya miatt ez a metódus érték-érték párokat ad eredményül.

Ez a két metódus a később tárgyalt `Map` objektumokkal való kompatibilitást szolgálja.

A következő példa egy lottószám generáló program. Ebben a halmazt arra használjuk, hogy kiküszöböljük a húzáskor a számok ismétlődését. A halmazban az elemek nem rendezettek és nincs is lehetőségünk rendezni őket. Ha mégis szeretnénk növekvő sorrendben megjeleníteni az értékeket, akkor az `Array.from()` metódussal tömbbé alakíthatjuk a halmazt, és a `sort()` metódusnál lévő példa szerint rendezhetjük.

```
let szamok = new Set();
while (szamok.size !== 5) {
  szamok.add(Math.floor(Math.random() * 90 + 1));
}
console.log(szamok);
```

10.2 Feladatok

- Írj programot, ami két halmazt (A és B) feltölt 20 db 1 és 100 közötti véletlenszámmal!
 - Írasd ki a két halmaz különbségét ($A \setminus B$)!
 - Írasd ki a két halmaz metszetét ($A \cap B$)!
 - Írasd ki a két halmaz egyesített halmazát ($A \cup B$)!
 - Döntsd el, hogy A halmaz részhalmaza-e B-nek ($A \subseteq B$)?
 - Kérj be egy 1 és 100 közötti számot, majd döntsd el, hogy eleme-e az A halmaznak ($x \in A$)!
- Készíts programot, amely összehasonlítja két halmazt, és eldönti, hogy azonosak-e!
- Írj programot, ami kiszámolja egy halmaz elemeinek összegét!

11 A Map adatszerkezet

A Map objektum kulcs érték párokat tárol, ahol a kulcs és az érték is lehet egyszerű és összetett adat, akár függvény vagy objektum is. Az adatok sorrendje rögzített, attól függ mikor szűrták be az adatot. Bár objektum típusú, az objektummal szemben iterálható.

Új Map-et az alábbi módon hozhatunk létre:

```
let aruk = new Map([
  ['alma', 300],
  ['paradicsom', 500],
  ['répa', 200]
]);
```

Másik lehetőség, amikor üres Map-et deklarálunk és utána töltjük fel adatokkal. Az adatok hozzáadására a set() metódust használjuk.

```
let aruk = new Map();
aruk.set('alma', 300);
aruk.set('paradicsom', 500);
aruk.set('répa', 200);
```

Ugyanez a metódus az adatok módosítására is alkalmas.

```
aruk.set('répa', 400);
```

A get() metódus a kulcs alapján visszaadja az értéket.

```
console.log(aruk.get('alma'));
```

A size tulajdonság megadja a Map méretét.

```
console.log(aruk.size);
```

A delete() metódus a megadott kulcsú elemet eltávolítja a Map-ből.

```
aruk.delete('paradicsom');
```

A clear() metódus a Map összes elemét törli.

```
aruk.clear();
```

A has() metódus igaz értékkel tér vissza, ha a paraméterként kapott kulcsú elem létezik.

```
console.log(aruk.has('alma'));
```

A forEach() a paramétereként megadott függvényt végrehajtja a Map minden elemén.

```
aruk.forEach(function(value, key){
  console.log(key + ' = ' + value);
});
```

Az entries() metódus egy iterátor objektumot ad vissza.

```
for (aru of aruk.entries()) {
  console.log(aru);
}
```

A `keys()` metódus a kulcsokat tartalmazó iterátor objektumot ad vissza.

```
for (aru of aruk.keys()) {  
    console.log(aru);  
}
```

A `values()` metódus az előzőhöz hasonlít, de aitt az értékek kerülnek visszadaásra.

```
for (aru of aruk.values()) {  
    console.log(aru);  
}
```

11.1 Feladatok

1. Hozz létre egy üres `Map`-et, majd adj hozzá néhány kulcs-érték párt.
2. Ellenőrizd, hogy egy adott kulcs már szerepel-e a `Map`-ben.
3. Számold meg, hány kulcs-érték pár van a `Map`-ben.
4. Távolíts el egy kulcs-érték párt a `Map`-ből.
5. Ellenőrizd, hogy a `Map` üres-e.
6. Hozz létre egy `Map`-et, amely szövegeket tartalmaz kulcsként és ezek hosszúságát értékül.
7. Írj egy függvényt, amely összegzi a `Map` értékeit (hosszúságokat) és visszaadja az összeget.
8. Adj hozzá egy kulcs-érték párt a `Map`-hez csak akkor, ha a kulcs még nem szerepel benne.
9. Módosítsd a `Map` egyik értékét, majd írd ki az új értéket.
10. Távolítsd el az összes kulcs-érték párt a `Map`-ből.
11. Hozz létre egy `Map`-et, amelyben az év hónapjai a kulcsok, és azok napjainak számossága az értékek.
12. Ellenőrizd, hogy egy adott évszak (tavasz, nyár, ősz, tél) szerepel-e a `Map` kulcsai között.
13. Hozz létre egy `Map`-et, amelyben a személyek nevei a kulcsok, és az életkoruk az értékek.
14. Hozz létre egy függvényt, amely visszaadja a `Map` legidősebb személyét és az ő életkorát.

12 Függvények

Mi a különbség a függvény és az eljárás között?

Nagyon leegyszerűsítve a legfontosabb különbség az, hogy az eljárásoknak nincs visszatérési értékük, míg a függvényeknek van. A programozásban nagyon gyakran használják ezt a két kifejezést szinonimaként, sőt még metódusként, szubrutinként vagy alprogramként is szoktak hivatkozni rájuk.

A függvény egy olyan programrész, mely egy külön alprogramnak tekinthető és általában a program más részei indítják el, vagyis hívják meg. A szoftver egyéb részeihez hasonlóan a függvény is műveletek meghatározott sorrendjét tartalmazza, melyet függvénytörzsnek nevezünk. Minden függvénynek átadhatóak értékek, ezek a paraméterek vagy argumentumok, és ő maga is visszaadhat értéket, amely a visszatérési érték lesz.

A JavaScriptben nincsenek eljárások. Minden függvény rendelkezik visszatérési értékkel, melyet a `return` kulcsszó után adhatunk meg. Amennyiben nem definiálunk ilyet, a függvény akkor is visszaad egy `undefined` értéket, vagyis mindig van valamilyen visszatérési érték!

A JavaScript függvényeinek tetszőleges számú paraméter átadható, melyeket a paraméterlistában adhatunk meg. Nem szükséges minden argumentumot kitölteni a függvény meghívásakor, ekkor a szabadon hagyott paraméterek a függvényen belül `undefined` értékűek lesznek.

A JavaScriptben a függvények az Objekt típus példányai, ezért úgy viselkednek, mint bármelyik más objektum. A JavaScript függvények különlegessége, hogy úgynevezett *first-class* tulajdonsággal rendelkeznek, más szóval:

- Futási időben létrehozhatóak.
- Ők maguk is objektumok, a `Function` objektumtól származnak.
- Lehetnek saját *property*-jeik, sőt metódusaik is, melyek referenciát tartalmaznak a konstruktorukra.
- Eltárolhatóak változóban.
- Átadhatóak más függvényeknek paraméterként.
- Szerepelhetnek visszatérési értékként.

12.1 Függvények létrehozása deklarációval

Függvényeket a JavaScript-ben többféle módon is létrehozhatunk. Leggyakrabban deklaráljuk a függvényt. Ilyenkor kötelező azonosítót adni a függvénynek, amellyel később hivatkozhatunk rá. Ezután a függvényt a teljes programban elérhetjük.

```
function add(a, b) {  
    return a + b;  
}  
  
console.log(add(3, 2));
```

12.2 Függvény kifejezések

A függvények bárhol, bármikor létrehozhatóak kifejezés formában. Ekkor egy objektum-referenciaként jelennek meg, melyet tetszőlegesen eltárolhatunk, vagy tovább adhatunk. A függvénykifejezést a deklarációhoz hasonlóan hozhatjuk létre, azonban ekkor a név paraméter is opcionális:

```
let add = function anAddFunction(a, b) {  
  return a + b;  
}  
  
console.log(add(3, 2));
```

Sőt, egy függvény lehet név nélküli (*anonymous*) is:

```
let add = function(a, b) {  
  return a + b;  
}  
  
console.log(add(3, 2));
```

12.3 Nyíl függvények

A nyíl függvények (*arrow function*) az ES6-ban jelentek meg. Más programozási nyelvekben lambda függvénynek hívjuk ezt a szerkezetet.

Az úgynevezett *Fat Arrow* (`=>`) operátor segítségével kihagyhatjuk a `function` kulcsszót a kifejezésből. Az így létrehozott függvények mindig anonim függvények lesznek!

Egysoros metódusok esetén lehetőségünk van elhagyni a kapcsos zárójeleket – illetve ebben az esetben a nyíl operátor utáni kifejezés automatikusan implicit visszatérési érték is lesz, azaz nem szükséges kiírni a `return` kulcsszót.

```
let add = (a, b) => a + b;  
console.log(add(5, 3));  
  
let udv = () => "Hello!";  
console.log(udv());
```

12.4 A `Function` konstruktor

Ha a függvények létrehozásáról beszélünk, akkor meg kell említeni, hogy a többi típushoz hasonlóan, a függvény is létrehozható a neki megfelelő, `Function` nevű beépített objektummal. Ekkor lehetőségünk van string-ként átadni a függvénytörzset – így akár dinamikusan, futási időben generálhatjuk a függvény magát.

A `Function` objektum konstruktorként való alkalmazása nem ajánlott, hiszen rengeteg szoftver-karbantartásági problémát okozhat, nehezíti a kód olvashatóságát és a hibakeresést, ráadásul a böngészők nem tudják megfelelően optimalizálni az így létrehozott programrészt.

```
let add = Function("a", "b", "return a + b");  
console.log(add(3, 2));
```

12.5 Önmagát hívó függvények

Létrehozhatunk olyan függvényt is, amit azonnal meghívunk. Ez az önmagát hívó, másnéven önkidőldő függvény nem tévesztendő össze a rekurzióval.

Mivel a JavaScript-ben a függvény is kifejezés, így lehetőségünk van a definiálás helyén azonnal futtatni is. Jogos lehet a kérdés, hogy miért is lenne erre szükség – ha úgyis azonnal lefut a függvény, akkor mi értelme egy újabb programblokkal bonyolítani az életünket? A válasz a változók láthatóságában keresendő. Mivel a JavaScript függvény scope-ot használ, szükség van valamilyen megoldásra ahhoz, hogy elszeparálhassuk a privát, csak az aktuális

kódhoz tartozó változókat a globális névtértől, illetve a programkomponenseket egymástól. Az önkioldó függvényekkel a JavaScript-ben natívan nem létező névtereket emulálhatjuk a legegyszerűbb módon. A következő példában a függvényen kívül nem érhető el sem a `nev` változó, sem az `udv()` függvény.

```
(function() {  
  let nev = "Elemér",  
      udv = function() {  
    return "Szia! " + nev + " vagyok!";  
  }  
  console.log(udv());  
})();
```

12.6 Paraméterek átadása

A JavaScript-ben a primitív típusok (számok, string és logikai) érték szerint, az objektumok pedig referencia szerint adódnak át.

- Az érték szerint átadott paraméter eredeti értéke nem változik meg. Más memóriacímre mutat ugyanis az átadott változó és a függvény paramétere. Több programozási nyelvben is van lehetőség primitív típust referencia szerint átadni; a JavaScript-ben erre nincs lehetőség.
- A referencia szerint átadott paraméter értéke megváltozhat. Egészen pontosan: ha magát a referenciát változtatjuk a függvényben, akkor az eredeti érték nem változik meg, viszont ha az objektum egy adatmezőjét, akkor igen. Az objektumokról később még lesz szó részletesebben.

```
let személy = {nev: "Pista", kor: 45};  
let i = 5;  
  
function f(személy, i) {  
  személy.nev = "Sanyi";  
  i = 8;  
}  
  
console.log(személy.nev); // Pista  
console.log(i); // 5  
f(személy, i);  
console.log(személy.nev); // Sanyi  
console.log(i); // 5
```

Az `f` függvény mindkét paraméterét megváltoztatja.

- A szám (`i`) nem változik: a globális `i` változó más memóriaterületen van, mint az `f` függvény `i` paramétere.
- Az objektum (`személy`) értéke megváltozik. Az igaz, hogy a globális `személy` és a függvény paraméter `személy` mint referencia, más memóriaterületen vannak, de ugyanoda mutatnak. Ezért, ha az egyik megváltoztatja a referencia által mutatott memóriaterület értékét, azt a másik is látja.

Tehát a referencia által mutatott memóriaterület tartalmát meg tudjuk változtatni, magát a referenciát azonban nem!

```
let személy = {nev: "Pista", kor: 45};

function f(személy) {
  p = {nev: "Jóska", kor: 30};
}

console.log(személy.nev); // Pista
f(person);
console.log(személy.nev); // Pista
```

Hivatkozások esetén különbséget kell tennünk a hivatkozás maga (azaz a referencia) és a hivatkozott érték (a tartalom) között.

Az ES6 a függvényparaméterek tulajdonságait is hasznos lehetőségekkel bővíti. Bevezetésre került az alapértékek fogalma, mellyel egyes argumentumok értékét állíthatjuk be arra az esetre, ha a függvény meghívásánál nincs átadva neki megfelelő érték.

```
function személy(nev = 'Elemér', lakhely = {varos: 'Pajkaszeg'}) {
  return nev + ' lakhelye: ' + lakhely.varos;
}

console.log(személy());
console.log(személy('Béla'));
```

Az ES6 alapértelmezett paraméter-értéke ráadásul megengedi, hogy bármilyen kifejezést megadhassunk neki, így akár egy függvényhívás eredménye is lehet argumentum. További hasznos tulajdonsága, hogy az alapérték futási időben értékelődik ki, tehát az így átadott metódus csak akkor hívódik meg, ha tényleg szükség van rá!

ES6 kompatibilis futtatókörnyezetben lehetőségünk van változó hosszúságú paraméterlisták kezelésére is, az úgynevezett *rest parameter* használatával. A `...jegyek` definiálása esetén a `jegyek` változó egy tömb lesz, mely a függvény nem nevesített paramétereit tartalmazza.

```
function tanulo(nev, ...jegyek) {
  return nev + ' jegyei: ' + jegyek.join(', ');
}

console.log(tanulo('Béla', 5, 4, 2, 5, 1, 3));
```

Az argumentumokat az `arguments` tulajdonság változón keresztül is elérhetjük. Ez a változó is biztosít lehetőséget változó hosszúságú paraméterlisták kezelésére. Ez viszont nem teljes értékű tömbként működik és a nevesített paraméterekkel nem foglalkozik. Az ES6-tól kezdve használata nem javasolt.

```
function maxKeres() {
  let max = -Infinity;
  for (let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}

console.log(maxKeres(52, 41, 26, 59, 12, 34));
```


12.7 Hoisting

A változónál bemutatott hoisting függvényekre is érvényes: a függvények – akárhol is vannak ténylegesen – felkerülnek a fejlécbe. Így meg tudjuk hívni azokat a függvényeket is, amelyeket később definiálunk:

```
console.log(add(3, 2));

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

12.8 Callback függvények

Callback-nek nevezzük azokat a függvényeket, amelyeket egy későbbi hívás kedvéért eltárolunk – e hívás pedig akkor történik meg, ha a *callback* számára megfelelő utasítássor lefutott. Tipikusan aszinkron hívások állapotváltozásaira kötjük őket (lásd: 20. fejezet), illetve különböző komponensek beállításainál találkozhatunk velük, hiszen segítségükkel a magunk számára bővíthetjük azok funkcionalitását. A callback függvényeket paraméterként adjuk át, más függvényeknek.

Mivel a JavaScriptben nagyon egyszerű anonim függvényeket létrehozni, eltárolni vagy éppen átadni, így a *callback*-ek írása a nyelv egy meghatározó eleme, nagyon sok helyen találkozhatunk vele.

Az alábbi példákban a *vegrehajt* függvény, az *muvelet* pedig változó; azok bármik lehetnek. (A *let*, a *function* és a *return* kulcsszavak.)

```
let osszeg = function(a, b) {
  return a + b;
}

let szorzat = function(a, b) {
  return a * b;
}

function vegrehajt(a, b, muvelet) {
  return muvelet(a, b);
}
console.log(vegrehajt(3, 2, osszeg));
console.log(vegrehajt(3, 2, szorzat));
```

Itt tehát az *osszeg* és a *szorzat* függvényt adjuk át paraméterként; a tényleges műveletet a *muvelet* hajtja végre. A példában az *osszeg* és a *szorzat* függvények a callback függvények.

Ez a korábban bemutatott szintaxissal is működik:

```
function osszeg(a, b) {
  return a + b;
}
function szorzat(a, b) {
  return a * b;
}
function vegrehajt(a, b, muvelet) {
  return operation(a, b);
}
console.log(vegrehajt(3, 2, osszeg));
console.log(vegrehajt(3, 2, szorzat));
```

Van egy másik, szintén gyakori szintaxis: miért adnánk egyáltalán nevet a függvénynek, vagy miért hoznánk létre változót, amikor csak egyszer használjuk?

```
function vegrehajt(a, b, muvelet) {  
    return muvelet(a, b);  
}  
  
console.log(vegrehajt(3, 2, function(a, b) {  
    return a + b;  
}))  
console.log(vegrehajt(3, 2, function(a, b) {  
    return a * b;  
}))
```

A funkcionális programozásban a lambda függvény (JavaScriptben nyíl függvény) alapvető fontosságú. Egy paraméter esetén a paraméter listának a zárójelét se kell kitenni.

```
function vegrehajt(a, muvelet) {  
    return muvelet(a);  
}  
  
console.log(vegrehajt(3, x => x + 1));  
console.log(vegrehajt(3, x => 2 * x));
```

A következő plédában egy tömb elemein végezzük el ugyanazt a műveletet egy callback függvény segítségével. A `tombMuvelet()` függvény egy tömböt és egy függvényt vár paraméterként, majd a tömb minden elemén végrehajtja a függvényt. Végül egy tömböt ad vissza eredményül.

```
let evek = [1954, 1990, 1963, 2000, 2010];  
  
function tombMuvelet(tomb, fgv) {  
    let eredmeny = [];  
    for (let e of tomb) {  
        eredmeny.push(fgv(e));  
    }  
    return eredmeny;  
}  
  
function korSzamitas(elem) {  
    return 2023 - elem;  
}  
  
let korok = tombMuvelet(evek, korSzamitas);  
console.log(korok);
```

12.9 Visszaadott függvények

A függvények nemcsak paraméterként kaphatnak másik függvényt, de a visszatérési értékük is lehet függvény. A példában az `interjuKerdes()` függvény paraméterként egy foglalkozást kap. A megadott foglalkozástól függően más-más függvénnyel tér vissza, amelyek paramétere egy név, visszatérési értékük pedig egy kérdés. A függvény változón keresztül és önállóan is meghívható. Utóbbi esetben két paramétert adunk meg. Elsőként a foglalkozást, másodikként a nevet.

```
function interjuKerdes(foglalkozas){
  if (foglalkozas === 'tanár'){
    return function(nev) {
      console.log(nev + ', meg tudná mondani, hogy milyen tárgyakat
oktat?');
    }
  } else if (foglalkozas === 'eladó'){
    return function(nev) {
      console.log(nev + ', hogyan kezelne egy vevői reklamációt?');
    }
  } else {
    return function(nev) {
      console.log('Mi a foglalkozása kedves ' + nev + '?');
    }
  }
}

let kerdesTanaroknak = interjuKerdes('tanár');
kerdesTanaroknak('Pál');

let kerdesEladoknak = interjuKerdes('eladó');
kerdesEladoknak('Katalin');
kerdesEladoknak('Géza');
kerdesEladoknak('Ilona');

interjuKerdes('tanár')('Péter');
interjuKerdes('programozó')('Elemér');
```

12.10 Closure

Egy belső függvény mindig képes hozzáférni az őt tartalmazó külső függvény paramétereire és változóihoz, még azután is, hogy a külső függvény befejezte a futását.

```
function nyugdij(ev) {
  let szoveg = 'Nyugdíjig hátralévő évek száma: ';
  return function(születesiEv) {
    let datumObjektum = new Date();
    let aktualisEv = datumObjektum.getFullYear();
    let kor = aktualisEv - születesiEv;
    console.log(szoveg + (ev - kor));
  }
}

let nyugdijazasUSA = nyugdij(66);
nyugdijazasUSA(1971);
nyugdij(66)(1971);
```

Létrehozunk egy `nyugdij()` függvényt, ami egy névtelen függvénnyel tér vissza. Ezt a függvényt eltároljuk a `nyugdijazasUSA` változóban. Mikor ezen keresztül meghívjuk a belső függvényt, az hozzáfér a szöveg változóhoz, holott a `nyugdij()` függvény már rég befejezte a futását.

12.11 Generátorok

Generátorok segítségével iterátor jellegű függvényeket tudunk létrehozni. Példaként egy olyan módszert szeretnénk megvalósítani, ami a Fibonacci számokat számolja ki és adja vissza, egyesével. Természetesen nem szeretnénk minden egyes lépésben újraszámolni. A természetes megoldás az, hogy eltároljuk valahol (pl. globális változóban, vagy egy osztály

attribútumaiként) az aktuális két számot, és lekérdezéskor léptetjük. Ugyanezt valósítja meg elegáns formában a generátor. Generátort a `function*` kulcsszóval tudunk létrehozni, visszatérni pedig nem a `return`, hanem a `yield` kulcsszóval kell.

```
function* fibonacci() {
  let n1 = 0;
  let n2 = 1;
  while (true) {
    yield n1;
    let sum = n1 + n2;
    n1 = n2;
    n2 = sum;
  }
}

const fib = fibonacci();
for (i = 0; i < 10; i++) {
  console.log(fib.next().value);
}
```

Ez a szintaktikai megoldás a funkcionális nyelvekben alapvető fontosságú.

12.12 Rekurzió

Az önmagukat hívó függvényeket rekurzív függvénynek nevezzük. Van közvetlen és közvetett rekurzió. Az első esetben az `A()` függvény tartalmaz egy `A()` függvényre, vagyis önmagára történő hívást. A második eset, amikor `A()` függvényben meghívjuk a `B()` függvényt, `B()` függvényben viszont az `A()` függvényt hívjuk meg.

A rekurzív függvénynek tartalmaznia kell egy alapesetet, aminek elérése a rekurzió befejezését jelenti, valamint egy általános esetet, ami a rekurzív hívást tartalmazza. A függvény ismétlődő meghívása minden esetben közelebb kell hogy vigyen az alapesethez.

A többi programozási nyelvhez hasonlóan a JavaScriptben is megvalósíthatunk önmagukat rekurzívan hívó függvényeket. Az alábbi példa a Fibonacci sorozat n -edik elemét adja vissza.

```
function fibonacci(n) {
  if (n < 2) {
    return n;
  } else {
    return fibonacci(n - 1) + fibonacci(n - 2);
  }
}

console.log(fibonacci(8)); // 21
```

A Fibonacci sorozat elemeit úgy képezzük, hogy a 0. eleme 0, az 1. eleme 1. A sorozat további elemeit az előző két elem összegeként képezzük. Ezért a 2. elem a $0+1=1$, a 3. $1+1=2$ és így tovább. (0, 1, 1, 2, 3, 5, 8, 13, 21...)

Az alapeset amikor $n < 2$. Ez lehet 0 vagy 1. Ekkor a visszatérési érték n . Az általános eset, amikor a függvény önmagát hívja: `fibonacci(n - 1) + fibonacci(n - 2)`.

12.13 Feladatok

1. Készíts függvényt, ami egy számot vár paraméterként és igaz értéket ad vissza, ha a szám páros!
2. Készíts egy függvényt, ami két szám közül kiírja a nagyobbbat.
3. Készíts egy függvényt, amely meghatározza az 1 és 100 közötti páros számok összegét.
4. Írj egy függvényt, amely kiszámolja egy szám számjegyeinek összegét.
5. Készíts egy függvényt, ami meghatározza egy számokat tartalmazó tömb legnagyobb elemét.
6. Írj egy függvényt, ami egy szöveget visszafelé ír.
7. Készíts egy függvényt, ami kiírja a számokat 1-től 100-ig, de a hárommal osztható számok helyett "Fizz"-t, az öttel osztható számok helyett "Buzz"-t, és a mindkettővel osztható számok helyett "FizzBuzz"-t ír ki.
8. Írj egy függvényt, ami egy adott másodpercszámot átalakít percek és másodpercek kombinációjává (pl. 90 másodperc = 1 perc 30 másodperc).
9. Készíts egy függvényt, amely két számot vár paraméterként, majd a számok szorzatát adja vissza.
10. Írj egy függvényt, amely eldönti, hogy egy adott szám prímszám-e.
11. Készíts egy függvényt, amely kiszámolja egy számokat tartalmazó tömb átlagát.
12. Írj egy függvényt, ami növekvő sorrendbe rendezi egy számokat tartalmazó tömb elemeit.
13. Készíts egy függvényt, amely összeadja egy számokat tartalmazó tömb elemeit.
14. Írj egy függvényt, amely kiszámolja egy adott szám faktoriálisát.
15. Készíts egy függvényt, amely kiszámolja egy számokat tartalmazó tömb elemeinek négyzetét.
16. Írj egy függvényt, amely kiszámolja, hány karakter van egy szóban vagy mondatban.

13 Objektumorientáltság

Hosszú fejlődésen ment keresztül az objektumorientáltság a JavaScript-ben. Objektumokat valójában már elég régóta létre lehet hozni, viszont ezek inkább hasonlítanak asszociatív tömbökre, azaz kulcs-érték párokra, metódus hozzáadási lehetőséggel, mint az objektumorientált programozásban megszokott objektumokra.

A későbbiekben a fejlődés az objektumorientált irányba mutatott. A konstruktor függvények megjelenésével már példányosítani is lehet, valamint külön kulcsszóval tudunk lekérdezőket (*getter*) és beállítókat (*setter*) létrehozni, valamint statikus metódusokat is létrehozhatunk.

Az ES6-ban jelent meg az a módszer, amit klasszikus értelemben objektumorientált programozásnak hívunk. Itt már van öröklődés is, metódus felülírás (*overriding*), az osztályon belül létrehozhatunk konstruktort.

13.1 Objektumok létrehozása

A kezdetleges objektumok valahogy így néztek ki:

```
let person = {
  firstName: "Csaba",
  lastName: "Faragó",
  age: 43
};

console.log(person.firstName);
console.log(person.lastName);
console.log(person.age);
```

Tulajdonságot utólag is adhatunk hozzá, ill. módosíthatjuk a már meglevőt:

```
person.country = "Hungary";
```

Ennek ebben a formában nincs sok köze az objektumorientáltsághoz, viszont – ahogy látni fogjuk – egész jól használható asszociatív tömbként.

13.2 Az `Object` osztály

Az alábbi példa a fentivel ekvivalens:

```
let person = new Object();
person.firstName = "Csaba";
person.lastName = "Faragó";
person.age = 43;
```

Itt tehát létrehozunk egy üres objektumot (asszociatív tömböt) a `new` kulcsszóval, és utána feltöltjük adatokkal.

13.3 Tulajdonságok

A tulajdonságokat háromféleképpen adhatjuk meg, ill. hivatkozhatunk rájuk:

```
person.firstName = "Csaba";
person["lastName"] = "Faragó";
let ageStr = "age";
person[ageStr] = 43;
```

- `person.firstName`: ez az objektumorientált programozásban megszokott mezőnév hivatkozás.
- `person["lastName"]`: ez inkább hasonlít az asszociatív tömb kulcsára.
- `person[ageStr]`: ha a mezőnév egy – akár generált – stringben adott, akkor ilyen módszerrel tudunk csak hivatkozni. Az objektumorientált világban stringben adott mezőnév esetén csak reflexiót használhatnánk, de az asszociatív tömbös megadási módszernél ez is működik.

13.4 Metódusok

A JavaScript objektumok abban különböznek az asszociatív tömböktől, hogy ezek tartalmazhatnak metódusokat is.

```
let person = {
  firstName: "Csaba",
  lastName: "Faragó",
  age: 43,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

Egy másik megadási mód, utólag adjuk hozzá a metódust.

```
person.fullName = function() {
  return this.firstName + " " + this.lastName;
}
```

13.5 Objektumok megjelenítése

Az objektum adatait többféleképpen megjeleníthetjük.

Mezőnévre történő hivatkozással, pl. `person.firstName`, ahogy a példákban is láthattuk.

A tulajdonságokon történő végigiterálással, a `for ... in` struktúra segítségével.

```
for (property in person) {
  console.log(property + ":" + person[property]);
}
```

Ha az objektum tartalmaz metódust, akkor ez a metódus törzsét írja ki.

Az `Object` osztály használatával, a kulcsokat, az értékeket ill. a kulcs-érték párokat tudjuk tömbbé (ill. ez utóbbit két dimenziós tömbök tömbjévé) alakítani, majd kiírni az alábbi módon:

```
console.log(Object.keys(person));
console.log(Object.values(person));
console.log(Object.entries(person));
```

13.6 Lekérdezők és beállítók

Az ES5-ben megjelentek a beállítók (*setter*) és lekérdezők (*getter*), a `set` ill. a `get` kulcsszavak használatával.

```
let person = {
  _firstName: "",
  _lastName: "",
  _age: 0,
  get firstName() {
    return this._firstName;
  },
  set firstName(firstName) {
    this._firstName = firstName;
  },
  get lastName() {
    return this._lastName;
  },
  set lastName(lastName) {
    this._lastName = lastName;
  },
  get age() {
    return this._age;
  },
  set age(age) {
    this._age = age;
  }
}
```

A mezőket tehát nem közvetlenül állítjuk be, hanem a beállítókon keresztül, függvényhívással. Adódik persze a kérdés, hogy ez mire jó, hiszen ugyanaz az eredménye, mint az eredetinek, viszont sokkal hosszabb, valójában áttekinthetlenebb is, ráadásul a mező neve nem egyezhet meg a setter nevével. Ebben a példában valóban nincs sok értelme, viszont ez egy fontos lépés az objektumorientáltság irányába. Az objektumorientált világban ugyanis az attribútumok (JavaScript terminológiával: azok a tulajdonságok, amelyek nem függvények) privátok, és azokat csak publikus metódusokon keresztül érhetjük el. Azokat a metódusokat, amelyek csak beállítják az attribútum értékét, beállítóknak (*setter*), azokat pedig, amelyek lekérdezik, lekérdezőknek (*getter*) nevezzük. Ezek a metódusok ugyanakkor más feladatot is elláthatnak, azon kívül, hogy csak beállítják ill. visszaadják az aktuális értéket, pl.:

- **Adat ellenőrzés:** például az életkor esetében hibát írhatnak ki, ha az érték negatív, és figyelmeztetést, ha az érték 120-nál magasabb.
- **Adat másolás:** ha az attribútum egy objektum, és lekérdezésnél csak úgy visszaadjuk, akkor a hívó fél meg tudja változtatni azt a példányt, amire az objektumunk hivatkozik. Ez megsérti az egységbe záras (*encapsulation*) objektumorientált szabályt, ami szerint egy adott objektumnak az attribútumai által leírt belső állapotát kívülről közvetlenül megváltoztatni sem szabad. Tehát pl. egy tömb közvetlen visszaadása helyett a lekérdező lemásolhatja a tömböt, és a másolatot adhatja vissza.
- **Statisztika készítése:** pl. meg lehet számolni, hogy hányszor történik tényleges lekérdezés ill. beállítás.
- **Hibakeresés felgyorsítása:** ha tudjuk, hogy a beállításnak mindenképpen át kell futnia a beállítón, akkor a hibakereséskor elég megszerezni annak az egy utasításnak a hívási láncát; nem kell az összes létező beállítást figyelni; ez utóbbi lehetetlen küldetés akkor, ha a beállítást egy olyan külső komponens teszi, aminek még a forráskódja sem áll rendelkezésre.

Jelenleg a JavaScript-ben mindez egy eléggé hibrid állapotban van, ugyanis addig, amíg nem vezetik be a privát mezőket, addig a fentieknek nincs sok értelmük. Már van egy javaslat a privát attribútumok szintaxisára: a mezőnév elé a `#` karaktert kell tenni, pl. `#firstName`. Egyenlőre a böngészők ezt még nem támogatják; ezáltal valójában az itt leírtak inkább érdekesek, mint hasznosak.

13.7 A függvények, mint objektumok

A JavaScript-ben a függvények ugyanúgy viselkednek, mint az objektumok. Egy függvénynek lehetnek tulajdonságai és metódusai is. Az ilyen függvényeket konstruktor függvényeknek nevezzük, és a szokásos függvény elnevezési konvenciótól eltérően itt nagybetűvel írjuk a függvények nevét.

```
function MyHello(name) {
  this.name = name;

  this.sayHello = function() {
    return `Hello, ${this.name}!`;
  }
}

let myHello = new MyHello("Csaba");
console.log(myHello.sayHello());
```

13.8 A this jelentése

A JavaScriptben a `this` több különböző jelentést is felvehet, attól függően, hogy hol hívjuk meg. Alapvetően a `this` arra az objektumra utal, aminek a kontextusában éppen meghívjuk, azonban lehetőségünk van ezt a viselkedést befolyásolni.

A `this` kulcsszót önmagában használva a globális objektumra mutat. Ez a böngészőablakban a window objektum.

```
console.log(this);
```

Ha függvényben használjuk, szintén a globális objektumra mutat.

```
function fgv() {
  return this;
}

console.log(fgv());
```

Objektum metódusában az objektumra mutat. Az alábbi példában a „Metódusok” fejezetben használt `person` objektumra.

```
fullName: function() {
  return this.firstName + " " + this.lastName;
}
```

A JavaScript három különböző metódust biztosít arra, hogy saját magunk állíthassuk be a `this` jelentését – más szóval, mi szabhassuk meg a futási kontextusát.

13.8.1 `call()` és `apply()`

Mindkettő metódus egyszerűen meghívja a függvényt, azonban az első paraméterükben megadható, hogy mi legyen a futási környezet, vagyis a `this` mire mutasson. A különbség

közöttük a meghívandó függvény paramétereinek átadásában van: míg a `call` paraméterlistája az eredeti függvényére hasonlít, azaz sorban, hagyományos paraméterként kell átadnunk a függvény argumentumait, addig az `apply` esetén az összes argumentumot egy tömbben adjuk át a második paraméter helyén.

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.fullName = function() {
    return this.firstName + " " + this.lastName;
  }
}

let p1 = new Person("Csaba", "Faragó", 43);
let p2 = new Person("László", "Nagy", 35);
console.log(p1.fullName.call(p2));
console.log(p1.fullName.apply(p2));
```

13.8.2 A `bind()` metódus

Sok esetben szükségünk van arra, hogy egy függvény (tipikusan *callback*) egy másik kontextusban fusson, de még ne hívódjon meg. Erre a problémára kínál elegáns megoldást a `bind` metódus, amely a paraméterül kapott objektumot a függvény kontextusaként fogja meghatározni úgy, hogy bármikor is hívjuk meg azt a megszokott módon, a kontextus mindig a beállított legyen.

```
const obj = {
  name: 'Example',
  logName: function() {
    console.log(this.name);
  }
};

const logNameFunction = obj.logName.bind(obj);

logNameFunction();
```

Ebben a példában a `bind()` metódust használjuk ahhoz, hogy a `logName` függvény `this` értékét az `obj` objektumra állítsuk be. Ezáltal a `logNameFunction` hívása ugyanazt eredményezi, mintha a `logName`-t az `obj` objektumon keresztül hívnánk meg.

Egy másik hasznos alkalmazása a `bind()`-nek a nyíl függvények helyettesítése, amelyek általában a környező kontextust veszik át:

```
const obj = {
  name: 'Example',
  logName: function() {
    setTimeout(function() {
      console.log(this.name);
    }.bind(this), 1000);
  }
};

obj.logName();
```

Ebben az esetben a `bind(this)` megoldást kínál arra, hogy az nyíl függvény ne veszítse el az `obj` objektum kontextusát a `setTimeout` hívásakor. A `bind()` használata tehát segít megoldani

a `this` kontextusával kapcsolatos problémákat és biztosítja, hogy a függvények a megfelelő kontextusban fussanak.

13.9 Konstruktorok

A függvényeknél már volt arról szó, hogy a JavaScript-ben a függvények és az osztályok nagyon közeli fogalmak. Pl. egy függvénynek is lehetnek "példány változói" (ami az objektumorientált attribútumnak felel meg), és tartalmazhat belső függvényeket is (ezek az objektumorientált metódusoknak megfelelő fogalom). Ha egy függvényt a `new` kulcsszóval hívjuk meg, akkor az már majdnem olyan, mintha osztályokat példányosítanánk, pl.:

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.fullName = function() {
    return this.firstName + " " + this.lastName;
  }
}

let p = new Person("Csaba", "Faragó", 43);
console.log(p.fullName());
```

Az így létre hozott konstruktor függvények esetén a konvenció szerint a függvény nevét nagy kezdőbetűvel írjuk. A megvalósítás kicsit szokatlan, de a kliens (hívó) oldalon már szinte teljesen olyan, mintha igazi objektumorientált megoldást látnánk.

Megjegyzés: az alap típusoknak is vannak konstruktoraik, pl. `new String()`, `new Boolean()`, de ezek használatát célszerű kerülni.

13.10 Prototípus

Amint fent láttuk, egy objektumhoz tudunk újabb tulajdonságokat rendelni. Adott konstruktorhoz viszont ugyanazzal a szintaxissal ezt nem tudjuk megtenni (folytatva a fenti példát):

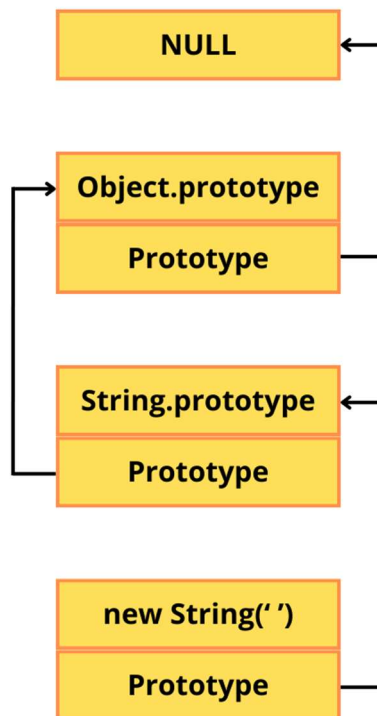
```
Person.country = "Hungary";
console.log(p.country); // undefined
```

A konstruktor függvényeknek viszont van egy ún. prototípusuk, amin keresztül már végre tudunk hajtani olyan változtatásokat, ami hatással van az összes példányra. Ezt a `prototype` kulcsszóval tudjuk megtenni:

```
Person.prototype.country = "Hungary";
console.log(p.country); // Hungary
```

13.11 Prototípus lánc

Minden objektumnak van egy `prototype` adattagja ami egy másik objektumra mutat. Ezt a másik objektumot hívjuk a szóbanforgó objektum prototípusának. Az objektum prototípusának is van egy prototípusa, és így tovább. Ezt nevezzük prototípus láncnak. A lánc végén a null objektumra mutató objektum van.



Prototípus lánc

13.12 Öröklődés

JavaScriptben az öröklődést hagyományosan a prototípus lánc kiterjesztésével oldjuk meg. Új objektum létrehozásakor az előbb leírt prototípus lánc végéhez adunk egy új elemet.

Felülírhatjuk az örökölt tulajdonságokat, vagy hozzáadhatunk újakat, magában az objektumban, illetve bárhol a prototípus láncban. A prototípus láncban való felülírás azt jelenti, hogy minden objektum, aminek a prototípus láncba tartozik a felülírt objektumot, szintén megkapja az új tulajdonságot. Ez a működés nagy rugalmasságot ad a nyelvnek, ugyanakkor rengeteg hiba okozója is lehet.

13.12.1 Object.create()

Az `Object.create()` segítségével könnyen megvalósíthatjuk az öröklődést (mellékhatások nélkül):

```

let Prototype = {
  name: 'Prototype',

  sayName: function () {
    console.log('Nevem: ' + this.name);
  }
};

Prototype.sayName();

let objektum = Object.create(Prototype);

objektum.name = 'objektum';
objektum.sayName();
  
```

Habár a kódunkban nem deklaráltuk az `objektum.sayName()` metódust, meghívásakor a motor kikeresi a `Prototype` objektumban megtalálhatót a prototípus láncban lépkedve.

Probléma lehet, hogy a `name` tulajdonságot külön sorban tudjuk csak felülírni. Ezt megoldhatjuk úgy, hogy átadunk egy `propertiesObject` objektumot az `Object.create()`-nek:

```
let objektum = Object.create(Prototype, {
  name: {
    value: 'objektum',

    enumerable: true,
    writable: true,
    configurable: true,
  }
});

objektum.sayName();
```

13.13 Osztályok

A felvezetőben volt arról szó, hogy az ES6-ban megjelentek az igazi objektumorientált nyelvi elemek a JavaScript-ben. Az alábbi példa tartalmaz osztály definíciót, konstruktort, attribútumokat, metódust, öröklődést és metódus felüldefiniálást is. A példában az osztálynak két attribútuma van: vezetéknév és keresztnév, és ebből képez teljes nevet. A magyar személy esetében előbb jön a vezetéknév, és utána a keresztnév.

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  fullName() {
    return this.firstName + " " + this.lastName;
  }
}

class HungarianPerson extends Person {
  constructor(firstName, lastName) {
    super(firstName, lastName);
  }

  fullName() {
    return this.lastName + " " + this.firstName;
  }
}

p1 = new Person("John", "Smith");
console.log(p1.fullName()); // John Smith

p2 = new HungarianPerson("Csaba", "Faragó"); // Faragó Csaba
console.log(p2.fullName());
```

13.14 Objektumszintű metódusok

JavaScriptben számos, az objektum egészét, és nemcsak egy-egy tulajdonságát befolyásoló funkció használatára van lehetőség. Ezek között vannak beállító és lekérdező műveletek.

- `Object.keys(obj)`: az objektum felsorolható tulajdonságainak nevét tartalmazó tömb.
- `Object.getOwnPropertyNames(obj)`: az objektum összes tulajdonságának nevét tartalmazó tömb.
- `Object.preventExtensions(obj)`: megakadályozza új tulajdonság hozzáadását az objektumhoz.
- `Object.isExtensible(obj)`: a fenti tulajdonság lekérdezése.
- `Object.seal(obj)`: a tulajdonságok és leírók attribútumának változtatását tiltja le, kivéve az értékek megváltoztatását.
- `Object.isSealed(obj)`: a fenti tulajdonság lekérdezése.
- `Object.freeze(obj)`: az objektum befagyasztása, azaz semmiféle változtatás sem engedélyezett az objektumon.
- `Object.isFrozen(obj)`: a fenti tulajdonság lekérdezése.

13.15 Feladatok

1. Hozz létre egy osztályt, amely egy autót reprezentál.
 - a. Az autónak legyenek tulajdonságai, például modell, évjárat és szín.
 - b. Készíts egy új osztályt, amely a fenti autó osztályból származik, és kiegészíti azt további tulajdonságokkal, például fogyasztás és üzemanyagtank méret.
 - c. Írj egy metódust az autó osztályban, amely kiszámolja az autó életkorát a gyártás éve és az aktuális év alapján.
 - d. Hozz létre két példányt az autó osztályból, és állítsd be az attribútumaikat különböző értékekre.
 - e. Hívd meg az egyik autó példányon a kor kiszámolására szolgáló metódust, majd írd ki az eredményt.
 - f. Az autó osztályhoz adj hozzá getter és setter metódusokat az egyes tulajdonságokhoz.
2. Írj osztályt, ami egy *Számítógép* objektumot valósít meg.
 - a. A számítógép tulajdonságai a következők legyenek: szabad memória MB-ban (szám), be van -e kapcsolva (logikai).
 - b. Készíts konstruktort az osztályhoz, ami minden tulajdonságot a paraméterlistából állít be, a konstruktornak legyenek alapértelmezett értékei, ami 1024 MB memóriával, kikapcsolva hozza létre a gépet.
 - c. Az osztálynak legyen egy *kapcsol* metódusa, ami nem vár paramétert. Ha a gép ki van kapcsolva, akkor kapcsolja be, egyébként kapcsolja ki.
 - d. Az osztálynak legyen egy logikai értékkel visszatérő *programMasol* metódusa, ami egy program méretét várja paraméternek MB-ban (szám). Ha a program ráfér még a gépre, és a gép be van kapcsolva, úgy csökkenjen a szabad memória a program méretével. A metódus térjen vissza igaz értékkel, ha sikeres volt a másolás.
 - e. Hozz létre két számítógépet a fenti konstruktorral. Az egyik az alapértelmezett értékeket használja, a másikat tetszőleges értékekkel hozd létre! Mindkét gép kikapcsolt állapotban kezdjen. Az alapértelmezett gépet kapcsold be, és másold rá először 800 MB, aztán 400 MB programot. A másik gépre másolj 1 MB programot. A másolások eredményeit írd ki.

3. Írj osztályt, ami egy Harcos objektumot valósít meg.
 - a. Az ember adattagjai a következők legyenek: név (String), életerő (int), harci erő (int). Az adattagok csak ebből az osztályból legyen elérhetőek.
 - b. Készíts az osztályhoz konstruktor, ami paraméterek alapján állítja be az adattagokat.
 - c. Az osztálynak legyen egy boolean értékkel visszatérő harcol metódusa, ami egy másik harcost kap paraméternek. A metódus mindkét harcos életerejét csökkentse a másik harcos harci erejével. Ha valamelyik harcos elveszti a harcot (életeroje 0 alá csökkenne), a metódus térjen vissza igazgal, egyébként hamissal.
 - d. Készítsd el a megfelelő metódusokat az adattagok lekérdezéséhez és módosításához.
 - e. létre két harcos objektumot. Harcoljanak, amíg valamelyikük el nem veszti a harcot.
4. Készítsünk egy „Háromszög” nevű osztályt, amely háromszögek kezelését valósítja meg. Az osztály a következő jellemzőkkel rendelkezzen:
 - a. Legyen képes tárolni a 3 csúcs koordinátáit
 - b. Legyen képes az adatok alapján eldönteni, hogy a háromszög egyenlőszárú, illetve szabályos-e
 - c. Legyen képes kiszámítani a saját területét és területét
 - d. Készítsünk továbbá egy másik osztályt, amelynek segítségével tesztelni lehet a Háromszög osztályt.
5. Készítsük el egy valós környezet szimulációját, melyben egy cég 5 db autóját kell működtetni az alábbi szabályok szerint:
 - a. Minden autónak van neve, a sofőrnek van valamennyi pénz, az autónak van fogyasztása, a benzintankjában bizonyos mennyiségű benzin, valamint megadható, hogy melyik autónak mekkora a benzintankja
 - b. A cégnek van egy benzinkútja, ahol tankolni lehet. A benzinkúton adott mennyiségű benzint tárol. A benzinnek ára van (literenként), és a benzinkútnak van kasszája is, ahova fizetni kell a tankolásért.
 - c. A cég főnöke az autókat elküldi adott hosszúságú útra.
 - d. A kiválasztott autó elindul, ha van elég benzinje, ellenkező esetben beáll tankolni.
 - e. Az autó akkor tud tankolni, ha a sofőrnek van elég pénze és a benzinkútban van elég benzin.
 - f. Ha az autó nem tud eljutni a kívánt távolságra, és nem tud eleget tankolni (nincs pénz, vagy nincs benzin a kútban), akkor az autó álljon le.
 - g. A szimuláció akkor ér véget, ha minden autó leállt.

14 Reguláris kifejezések

14.1 Reguláris kifejezések létrehozása

Reguláris kifejezésekkel szövegkeresési mintákat írhatunk le, ezen minták karakterláncokban karakterkombinációk megfeleltetésére használhatók. Egyik leggyakoribb felhasználása az űrlapok beviteli mezőinek ellenőrzése; például megvizsgálhatjuk, hogy a megadott e-mail cím formátuma megfelelő-e. A JS 1.2 és későbbi verziókban van támogatás reguláris kifejezések használatára. JavaScriptben a reguláris kifejezések is objektumok. A RegExp beépített prototípus szolgál ezen kifejezések kezelésére, egy RegExp objektum egy megadott kifejezés mintáját tartalmazza. Rendelkezik olyan tulajdonságokkal és eljárásokkal, amelyek használatával karakterláncokban kereséseket és a találatok szövegcserejét is elvégezheti.

14.1.1 Létrehozás

Reguláris kifejezést kétféleképpen hozhatunk létre:

1. A RegExp konstruktorfüggvényével:

```
reg = new RegExp("minta", "kapcsolok")
```

(A kapcsolókról később még bővebben lesz szó.) A konstruktor használata futásidejű fordítást szolgáltat a reguláris kifejezés számára. Akkor érdemes használni, ha előre tudjuk, hogy a reguláris kifejezés mintája meg fog változni, vagy ha más forrásból érkezik, például felhasználói adatbevitelből, mivel az így megadott minta minden egyes hivatkozáskor újra kiértékelődik.

2. Objektum inicializáló használatával:

```
reg = /minta/[kapcsolok]
```

Ebben az esetben a reguláris kifejezés előfordítódik, ha a reguláris kifejezés változatlan marad, a jobb teljesítmény érdekében ezt a megoldást érdemes használni.

Egy reguláris kifejezés állhat egyszerű karakterekből, vagy egyszerű és speciális karakterek kombinációjából.

14.1.2 Egyszerű minta

Az egyszerű mintákat közvetlen megfeleltetésre használjuk. Például a `/men/` minta csak azoknak a karakterláncokban található karakterkombinációknak felel meg, ahol a "men" karakterek pontosan így, és ebben a sorrendben szerepelnek. Azaz, a megfeleltetés sikeres lesz a "Laci elment bevásárolni" karakterlánc esetén, viszont a "Nem engedem" karakterláncnál már nem, mivel ez nem tartalmazza pontosan a `/men/` szövegrészt.

14.1.3 Speciális karaktere

Természetesen gyakran a közvetlen megfeleltetésnél bonyolultabb keresési módokra van szükség, ezeket segítik a speciális karakterek. Például megadhatjuk, hogy az "m" után az "en" karakterek tetszőleges távolságra, de mindenképp benne legyenek a karakterláncban. Ennek megadása: `/m*en/`. A `*` jelenti azt, hogy az "m" karaktert 0 vagy több karakter követi, majd az "en" karakterek. Ekkor a "Nem engedem" szövegnél is sikeres lesz a megfeleltetés.

A következő táblázatban a reguláris kifejezésekben előfordulható speciális karakterek vannak felsorolva.

karakter	jelentése
\	Vagy azt jelzi, hogy a következő karakter különleges és ne legyen betű szerint értelmezve (például az <code>s</code> karakterből a <code>\s</code> üres karaktert jelentő különleges karaktert hoz létre), vagy egy különleges karaktert alakít át úgy, hogy betű szerint legyen értelmezve (például a <code>*</code> a <code>*</code> karaktert jelenti)
^	sor elejét egyezteteti
\$	sor végét egyezteteti
*	a megelőző karakter 0 vagy többszöri előfordulását írja elő
+	a megelőző karakter 1 vagy többszöri előfordulását írja elő
?	a megelőző karakter 0 vagy egyszeri előfordulását írja elő
.	tetszőleges karakter egyszeri előfordulása
{ }	a megelőző karakter többszöri ismétlődését írja elő <code>{n}</code> pontosan <code>n</code> db előfordulás <code>{n,m}</code> <code>n</code> és <code>m</code> közötti előfordulás <code>{n, }</code> <code>n</code> vagy többszöri előfordulás
()	a zárójelben lévő mintának megfelelő szöveget megjegyzi, lehetővé téve a későbbi felhasználását
[]	a zárójelben megadott karakterhalmaz egy elemének előfordulását írja elő <code>[n,m]</code> <code>n</code> vagy <code>m</code> bármelyike <code>[n-m]</code> <code>n</code> és <code>m</code> közötti tartomány bármelyike <code>[^n,m]</code> inverz; bármi, csak nem <code>n</code> vagy <code>m</code>
	a két oldalán álló minták egyikének előfordulását írja elő (“vagy”)
[b]	visszatörlés karakter előfordulását írja elő
\cX	kontrollkarakter előfordulását írja elő
\d	számjegy előfordulását írja elő, megegyezik a következővel: <code>[0-9]</code>
\D	az előző tagadása, azaz <code>[^0-9]</code>
\n	soremelés előfordulását írja elő
\f	lapdobás előfordulását írja elő
\t	tabulátor előfordulását írja elő
\v	vertikális tabulátor előfordulását írja elő
\r	kocsivissza előfordulását írja elő
\s	üres karakter előfordulását írja elő, azaz <code>[\f\t\v\r\n]</code>
\S	az előző tagadása, azaz <code>[^ \f\t\v\r\n]</code>
\w	betű, számjegy vagy aláhúzás, azaz <code>[0-9a-zA-Z_]</code>
\W	az előző tagadása, azaz <code>[^0-9a-zA-Z_]</code>

14.1.4 Kapcsolók

A reguláris kifejezéseknek van két opcionálisan használható kapcsolójuk, amelyek lehetővé teszik a globális és kis- és nagybetű-érzékeny keresést. A globális kereséshez a `g`, a kis- és nagybetű-érzékeny kereséshez az `i`, többsoros szövegek kereséséhez az `m` kapcsoló használható. Ezek a kapcsolók használhatók külön, vagy együtt. A kapcsolók a reguláris kifejezés részei, később el nem vehetők, és hozzá sem adhatók.

14.2 Reguláris kifejezések használata

Reguláris kifejezéseket a `RegExp` objektum `test` és `exec` metódusai, valamint a `String` objektum `match`, `replace`, `search`, és `split` metódusai használnak.

14.2.1 A `test()` metódus

Letesztel egy egyezést a karakterláncon, visszatérési értéke `true` vagy `false`.

```
let reg = /^.+@.+\\.\\{2,6}\\$/;
console.log(reg.test("kati@mail.hu"))
```

Megvizsgálja, hogy a paraméterként kapott string e-mail formátumú-e.

14.2.2 Az `exec()` metódus

Keresést hajt végre egy karakterláncon, visszatérési értéke egy eredménytömb.

```
let reg = /a(b+)a/g;
console.log(reg.exec("cabbabsbz"));
```

A 'aba', 'abba', 'abbba', stb. minták keresése a kapott stringben.

14.2.3 A `match()` metódus

Keresést hajt végre egy karakterláncon, visszatérési értéke egy eredménytömb, vagy ha nincs egyezés, null.

```
let str="88sd433 p23ko";
console.log(str.match(/\d+\s/g));
```

Olyan mintát keres, ahol egy vagy több számjegyet üres karakter követ.

14.2.4 A `replace()` metódus

Keresést hajt végre egy karakterláncon, és az egyező szövegrészeket lecseréli az adott karakterláncra.

```
let str="hehe hee hee";
console.log(str.replace(/e+/g, "a"));
```

Eredménye: "haha ha ha".

14.2.5 A `search()` metódus

Letesztel egy egyezést egy karakterláncon, visszatérési értéke az egyezés indexe, vagy -1, ha nincs egyezés.

```
let str = "abbabsbz";
console.log(str.search(/^a(b+)a/g));
```

Eredménye: 0. Mivel a reguláris kifejezés `^`-vel kezdődik, csak a string elején lehet egyezés.

14.2.6 A `split()` metódus

Egy reguláris kifejezést vagy egy karakterláncot használ egy adott karakterlánc tördelésére és tömbbe helyezésére.

```
let str = "alma körte;barack";
console.log(str.split(/\s|;/));
```

A pontosvesszővel vagy üres karakterrel elválasztott karakterláncokat a tömbbe helyezi.

14.3 Zárójelezett szövegrész megfeleltetése

Egy reguláris kifejezés mintában foglalt zárójelpár hatására a megfelelő egyezésrész megjegyződik. Például, az `/a(b)c/` megfelel az 'abc' karaktereknek és a 'b' megjegyződik. A megjegyzett zárójelezett szövegrész-megfeleltetések visszahívásához a `RegExp` objektum `$1`, ..., `$9` tulajdonságai, vagy az `Array` `[1]`, ..., `[n]` elemek használhatóak. A következő szkript a `replace` metódust használja karakterláncban szavak cseréjéhez. A kicserélendő szöveghez a szkript a `$1` és `$2` tulajdonságok értékeit használja.

```
let reg = /(\w+)\s(\w+)/;
let str = "Kiss Zsuzsa";
let newstr = str.replace(reg, "$2, $1");
console.log(newstr);
```

Eredménye: "Zsuzsa, Kiss"

14.4 Példák a RegEx alkalmazására

14.4.1 Dátum feldolgozás

```
[0-9]{4}[^0-9]*[0-9]{2}[^0-9]*[0-9]{2}
```

Hogy is olvasandó a példa? `[0-9]{4}` négy darab számjegy (év); amelyet követ `[^0-9]*` nulla vagy akár több nem számjegy karakter (esetleges elválasztó); majd `[0-9]{2}` két számjegy (hónap); utána ismét `[^0-9]*` jöhet elválasztó; és végül `[0-9]{2}` két újabb számjegy.

Ezzel még csak félmunkát végeztünk. Tudunk azonosítani egy dátumot, de nem tudunk hivatkozni az egyes tagokra. A kerekzárójellel emeljük ki azokat a csoportokat amik számunkra érdekes adatokat hordoznak.

```
([0-9]{4})[^0-9]*([0-9]{2})[^0-9]*([0-9]{2})
```

A visszkapott tömb tartalmazza a dátumot egészben és részekre bontva.

```
let reg = /([0-9]{4})[0-9]*([0-9]{2})[0-9]*([0-9]{2})/;
let datum = "2006. 07. 22.";
console.log(reg.exec(datum));
```

A tömb nullás sorszámu rekeszébe kerül az egész kifejezésnek megfelelő minta (látható, hogy a végső pont nincs benne, hiszen a keresett kifejezésünk véget ér a napot jelölő két számjeggyel). Az egyes sorszámtól kezdve a tömbbe kerültek az általunk (kerekzárójellel) megjelölt csoportok. A kiindulási dátum lehetett volna "2006-07-22" vagy "20060722" is, a kifejezés mindet megfejtí.

Bizonyos dátumformátumok nem pótolják ki kétszámjegyre az adatokat. Például "2006/7/22". Ilyenkor az elválasztók jelenléte a döntő. Az elválasztók között mindig van egy vagy több számjegy:

```
([0-9]{4})[0-9]*([0-9]+)[0-9]*([0-9]+)
```

14.4.2 Bankszámlaszám ellenőrzése

```
^[0-9]{8}([ -]?[0-9]{8}){1,2}$
```

A bankrendszer behatóbb ismerete nélkül feltételezem, hogy egy számlaszám 16 vagy 24 számból áll, és nyolcasával szokás öt elválasztani.

Értelmezzük a kifejezést: nyolc számjegy; majd következik (kérdőjel, azaz 0 vagy 1 darab előfordulás) egy elválasztó karakter (szóköz vagy kötőjel); ezután ismét nyolc számjegy. Az utolsó két kifejezésből létrehozunk egy csoportot (kiemelt rész – kerek zárójel), amiből legalább egy (16 számjegyre számlaszám) vagy legfeljebb kettő (24 számjegyre számlaszám) fordulhat elő. A kifejezés a start ^ jellel kezdődik és a vége \$ jellel fejeződik be, így teljes egyezést vizsgálunk, nem elég, ha a minta csak tartalmazza a számlaszámot (enélkül például a "abc12345678-12345678" is megoldás lenne).

```
<script>
function szamlaCheck() {
    let reg = /^[0-9]{8}([ -]?[0-9]{8}){1,2}$/;
    let s = document.getElementById('szamlaszam').value;
    alert(reg.test(s));
}
</script>
<input type="text" id="szamlaszam"/>
<input type="button" onclick="szamlaCheck();" value="Számlaszám?"/>
```

14.5 Feladatok

1. Ellenőrizd, hogy egy adott szöveg tartalmaz-e egy bizonyos karakterláncot.
2. Ellenőrizd, hogy egy szöveg a-z karaktereket tartalmaz-e.
3. Keresd meg és válaszd ki az összes e-mail címet egy szövegben.
4. Ellenőrizd, hogy egy szöveg csak számokat tartalmaz-e.
5. Keresd meg és számold meg az összes évszámot egy szövegben.
6. Ellenőrizd, hogy egy szöveg csak nagybetűket tartalmaz-e.
7. Keresd meg és válaszd ki az összes URL-t egy szövegben.
8. Ellenőrizd, hogy egy szöveg tartalmaz-e olyan szavakat, amelyek három vagy több magánhangzót tartalmaznak.
9. Keresd meg és válaszd ki az összes telefonszámot egy szövegben.
10. Ellenőrizd, hogy egy szöveg minden mondata pontosvesszővel végződik-e.
11. Keresd meg és válaszd ki az összes számot, amely legalább három számjegyből áll, egy szövegben.
12. Ellenőrizd, hogy egy szöveg csak betűket és számokat tartalmaz-e.
13. Keresd meg és válaszd ki az összes dátumot egy szövegben (például: YYYY-MM-DD formátumban).
14. Ellenőrizd, hogy egy szöveg csak speciális karaktereket tartalmaz-e.

15 Kivételkezelés

A JavaScript-ben a kivételkezelés szintaxisa sokban hasonlít a Java-ra. Lássunk egy példát!

```
function divide(a, b) {
  if (b == 0) {
    throw "division by zero";
  }
  return a / b;
}
function logDivide(a, b) {
  try {
    console.log(a + "/" + b + "=" + divide(a, b));
  } catch (error) {
    console.log("Error: " + error);
  } finally {
    console.log("Division finished.");
  }
}
logDivide(5, 0);
logDivide(5, 2);
```

A `divide` kivételt dob, ha az osztó 0. A `throw` utasítással tudunk saját kivételt létrehozni.

A hívó oldal a `try ... catch ... finally` struktúrával kezeli a kivételt. A `finally` mindenképp lefut, akár történt kivétel, akár nem. A fenti példában ennek nincs jelentősége, hiszen írhattuk volna a `try ... catch` után is. Összetettebb esetekben, tehát ha pl. kivétel váltódik ki a `catch` ágban, vagy nincs is `catch` ág, esetleg van valahol egy `return`, már van jelentősége.

Habár olyan szintű kivétel hierarchia a JavaScript-ben nincs, mint a Java-ban, itt is finomhangolni tudjuk a kivételeket az `Error` osztály segítségével. Lássunk erre is egy példát:

```
class DivisionByZeroError extends Error {
  constructor(...params) {
    super(...params);
  }
}
function divide(a, b) {
  if (b == 0) {
    throw new DivisionByZeroError();
  }
  return a / b;
}
function logDivide(a, b) {
  try {
    console.log(a + "/" + b + "=" + divide(a, b));
  } catch (error) {
    if (error instanceof DivisionByZeroError) {
      console.log("DivisionByZeroError");
    } else {
      console.log("Error: " + error);
    }
  } finally {
    console.log("Division finished.");
  }
}
logDivide(5, 0);
logDivide(5, 2);
```

Számos előre definiált `Error` osztály van: `EvalError`, `InternalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`.

16 Destrukturálás

A destrukturálás egy szintaktikai lehetőség, amivel a kódot tömörebbé, és kellő gyakorlattal olvashatóbbá tudjuk tenni.

Az egyik alapprobléma: tegyük fel, hogy van egy tömbünk, és annak az elemeit változókhoz szeretnénk rendelni:

```
let arr = [2, 3];
let a = arr[0];
let b = arr[1];
console.log(a);
console.log(b);
```

Kicsit bőbeszédű, az ES6-tól ezt így is írhatjuk:

```
let [a, b] = [2, 3];
console.log(a);
console.log(b);
```

Az ugyancsak ES6-ban megjelent kiterjesztés (*spread*) operátorral (...) is használható:

```
let [a, b, ...c] = [1, 2, 3, 4, 5];
console.log(a);
console.log(b);
console.log(c);
```

Ez esetben a `c` értéke `[3, 4, 5]` lesz.

Objektumokkal is működik, sőt, a gyakorlatban inkább ott használatos. Vegyük a következő példát:

```
let person = {
  firstName: "Csaba",
  lastName: "Faragó",
  age: 43
}

let vezetéknév = person.lastName;
let keresztnév = person.firstName;
console.log(vezetéknév);
console.log(keresztnév);
```

Az értékadás összevonható:

```
let {lastName: vezetéknév, firstName: keresztnév} = person;
```

Tehát a mezőnevekből tudja a JavaScript, hogy melyik új változónak mit kell értékül adni.

Most tegyük fel, hogy egy olyan függvényt szeretnénk írni, ami a személy keresztnévét és zárójelben az életkorát írja ki, ahogy egyes tévéműsorokban szokás. A hagyományos megoldás:

```
function printShort(person) {
  console.log(person.firstName + ' (' + person.age + ')');
}

printShort(person);
```

A destrukturált megoldás:

```
function printShort({firstName: firstName, age: age}) {  
  console.log(firstName + ' (' + age + ')');  
}
```

Ennek van egy továbbfejlesztett változata is, ami szintén kihasznál egy ES6-os újdonságot:

```
function printShort({firstName, age}) {  
  console.log(firstName + ' (' + age + ')');  
}
```

Elég tehát csak felsorolni a mezőneveket.

17 Eseménykezelés

Miközben a felhasználó böngészi a weboldalunkat, történhetnek különféle események – pl. a felhasználó rákattint egy oldalelemre, egy elem fölé viszi a kurzort, egy HTML elem betöltődik vagy megváltozik. Ezekhez az eseményekhez társíthatunk eseménykezelő függvényeket, amelyek akkor hívódnak meg, ha az adott esemény bekövetkezik.

Az eseménykezelés egyik módja, hogy az elemeknek adott, eseménykezeléssel kapcsolatos attribútumokkal szabályozzuk az események működését. A kérdéses HTML elemet ellátjuk az alábbi attribútumok valamelyikével, és az attribútum értékeként megadjuk az eseménykezelést végző függvényt.

17.1 A JavaScript eseményei

Az egérkezeléshez kapcsolódó események akkor következnek be, ha az egér valamilyen interakcióba kerül egy HTML elemmel.

Esemény	Mikor következik be
onclick	kattintunk az egérrel
ondblclick	duplán kattintunk az egérrel
onmousedown	lenyomjuk az egérgombot
onmouseup	felengedjük az egérgombot
onmousemove	mozgunk az egérrel az elem felett
onmouseenter onmouseover	az egérmutató egy elemre kerül
onmouseleave onmouseout	a mutató elhagyja az elemet
oncontextmenu	jobb egérgombbal az elemre kattintva megnyílik a helyi menü
onwheel	az egérgörgő gördül egy elem felett

Az elemek fókuszba kerülésekor vagy a fókusz elvesztésekor az alábbi események következhetnek be.

Esemény	Mikor következik be
onfocus onfocusin	egy elem fókuszba kerül
onblur onfocusout	egy elem elveszti a fókusz

A billentyűzettel kapcsolatos események.

Esemény	Mikor következik be
onkeypress	lenyomunk egy billentyűt
onkeydown	billentyű leütése
onkeyup	billentyű felengedése

A CSS animációk is válhatnak ki eseményeket

Esemény	Mikor következik be
onanimationstart	az animáció elindult
onanimationend	az animáció befejeződött
onanimationiteration	az animáció ismétlődik
ontransitionend	egy CSS transition befejeződött

Ha az elemeket „*drag and drop*” módszerrel mozgatjuk az alábbi eseménykezelőket használhatjuk.

Esemény	Mikor következik be
ondragstart	az elem húzása elkezdődött
ondrag	az elem húzása folyamatban van
ondragenter	a húzott elem a célterület fölé ért
ondragover	a húzott elem a célterület fölött van
ondragleave	a húzott elem elhagyta a célterületet
ondragend	az elem húzása befejeződött
ondrop	a húzott elem a célterületre került

A vágólap műveletekkel kapcsolatban is használhatunk eseménykezelőket.

Esemény	Mikor következik be
oncopy	egy elem tartalmát kimásoljuk
oncut	egy elem tartalmát kivágjuk
onpaste	egy elembe tartalmat illesztünk a vágólapról

Az érintőképernyő eseményei külön csoportot alkotnak. Ezek az események csak érintőképernyő használatakor működnek.

Esemény	Mikor következik be
ontouchstart	ujjunkkal megérintjük a képernyőt
ontouchmove	ujjunkat mozgatjuk a képernyőn
ontouchend	ujjunkat felemeljük a képernyőről
ontouchcancel	az érintés megszakad

A video és hangfájlok betöltésével és lejátszásával kapcsolatban is keletkeznek események.

Esemény	Mikor következik be
onloadstart	a média betöltése megkezdődött
ondurationchange	a média időtartama megváltozik (NaN-ról tényleges időtartamra)
onloadedmetadata	a metaadatok betöltődtek
ononloadeddata	betöltődött az első képkocka
onprogress	a böngésző hangot vagy videót tölt le
oncanplay	a böngésző megkezdte a média lejátszását
oncanplaythrough	a böngésző szerint a média megállás nélkül lejátszható
onplay	a média lejátszása elindul
onplaying	a média lejátszása elindul (szüneteltetés után is)
ontimeupdate	a média lejátszási pozíciója megváltozik
onended	a lejátszás véget ért
onabort	a média vagy fájl letöltése megszakad (nem hiba esetén)
onemptied	a hang vagy video üres
onerror	hiba történik a média vagy fájl letöltése közben
onpause	lejátszás szünetel
onratechange	a lejátszás sebessége megváltozik
onseeked	a felhasználó új pozíciót keres a médiában
onseeking	a felhasználó új pozíció keresését kezdi
onstalled	a böngésző nem létező médiaadatokat próbál letölteni
onsuspend	a böngésző nem kap médiaadatot
onvolumechange	a hangerőben változás történik
onwaiting	a lejátszó vár a következő képkockára

Űrlapokkal kapcsolatos események:

Esemény	Mikor következik be
onchange	egy form elemen változás következik be
oninput	egy <code><input></code> vagy <code><textarea></code> mező értéke megváltozik
oninvalid	egy beküldendő <code><input></code> elem érvénytelen
onreset	az űrlap resetelésre kerül
onsearch	egy <code><input type="search"></code> elemben keresést indítanak
onselect	a felhasználó kijelölt egy szöveget (input, textarea)
onsubmit	az űrlap elküldésre kerül

Nyomtatással, ablakkezeléssel kapcsolatos események.

Esemény	Mikor következik be
onbeforeprint	a nyomtatási párbeszédablak megnyitásakor
onafretpoint	a nyomtatási párbeszédablak bezárásakor
onbeforeunload	egy oldal elhagyása előtt
onfullscrenchange	az ablak teljesképernyős nézetre vált (böngészőfüggő)
onfullscreenerror	az ablak nem tud teljesképernyőre váltani (böngészőfüggő)
onhashchange	az URL # utáni része megváltozik
onload	egy objektum betöltődik
onmessage	üzenet érkezik egy külső objektumtól
onoffline	a böngésző offline üzemmódba kerül
ononline	a böngésző online üzemmódba kerül
onopen	egy külső forrással létrejön a kapcsolat
onpagehide	a felhasználó elnavigál egy oldalról
onpageshow	a felhasználó egy weboldalra navigál
onpopstate	az ablak előzményei megváltoznak
onresize	a böngészőablak átméretezésre kerül
onscroll	az elem gördítősávját görgetik
onshow	egy helyi menü megjelenik (csak firefoxban)
onstorage	a webstorage tárterülete változik
ontoggle	a felhasználó megnyitja vagy bezárja a <code><details></code> elemet
onunload	a felhasználó elnavigál, bezár vagy újratölt egy oldalt

Az eseménykezelőt leggyakrabban egy függvény hívására használjuk, ami lefut az esemény bekövetkezésekor.

```
<button onclick="katt()">Kattints!</button>
<script>
  function katt() {
    console.log('Megnyomtad a gombot!');
  }
</script>
```

Ha a program csak egysoros, akkor az alábbi megoldást is alkalmazhatjuk:

```
<button onclick="console.log('Megnyomtad a gombot!');">Kattints!</button>
```

17.2 Eseménykezelés az `addEventListener()` metódussal

Az előző példa kizárólag az attribútumokkal történő eseménykezelést mutatta be. Egy másik módszer az eseménykezelésre, ha egy DOM-beli elem `addEventListener()` metódusát használjuk, ezzel rendeljük hozzá az elemhez az eseménykezelő függvényt (ekkor az elemnek

nem kell semmilyen attribútumot adni). Egy HTML elemhez több eseménykezelő is hozzárendelhető (akár ugyanarra az eseménytípusra is).

Az `addEventListener()` metódus paraméterei sorban:

- az esemény neve, amit `on` előtag nélkül adunk meg (pl. `click`, `load`, `unload` stb.)
- eseménykezelő függvény, amely az esemény bekövetkezésekor hívódik meg
- elhagyható, logikai típusú paraméter, amely az eseménykezelő lefutásának időpontját szabályozza (bővebben lásd: "capturing és bubbling" doboz).

Példa: Eseménykezelő hozzárendelése egy gombhoz az `addEventListener()` metódussal.

```
<body>
  <button type="button" id="my-btn">Kattints rám!</button>
  <script>
    //gomb megkeresése
    const button = document.getElementById("my-btn");

    // kattintás esemény kezelése
    button.addEventListener("click", function() {
      alert("Hurrá, működik az eseménykezelő!");
    });
  </script>
</body>
```

Capturing és bubbling

Tegyük fel, hogy van egy HTML elemünk, ami egy másik HTML elembe van beágyazva! Mind a beágyazó elemhez, mind a beágyazott elemhez hozzárendelünk egy-egy eseménykezelőt ugyanarra az eseménytípusra – mondjuk a kattintásra. Ha a belső elemre kattintunk, akkor mindkét elem eseménykezelője meghívódik. Az `addEventListener()` metódus harmadik paraméterével szabályozhatjuk, hogy milyen sorrendben legyenek ezek meghívva.

Ha `true`-ra állítjuk a harmadik paramétert, akkor *capturing* történik. Ekkor az eseményt először a legkülső elem kezeli le, majd ezután mindig az eggyel "beljebb" található elem eseménykezelő függvénye hívódik meg.

Ha `false`-ra állítjuk a harmadik paramétert, akkor *bubbling* történik. Ekkor az eseményt először a legbelső elem kezeli le, majd ezután mindig az eggyel "kijebb" található elem eseménykezelő függvénye hívódik meg. Ha nem adjuk meg expliciten a 3. paraméter értékét, akkor alapértelmezett módon mindig bubbling történik.

Nézzünk egy példát! Figyeljük meg a konzolon, hogy az "eseménykezelők" feliratra kattintva, *capturing* esetén "kívülről befelé", míg *bubbling* esetén "belülről kifelé" sorrendben hívódnak meg az elemek eseménykezelői!

A két függvény, kiírja annak a HTML elemnek a nevét, amelyen az esemény bekövetkezett. A `capture()`-t *capturing*, a `bubble()`-t pedig *bubbling* módban fogjuk használni.

```
<body>
  <p id="my-paragraph">
    Példa <strong id="my-strong">eseménykezelők</strong> használatára.
  </p>
  <script>
    function capture() { console.log("Capturing: " + this.tagName); }
    function bubble() { console.log("Bubbling: " + this.tagName); }

    // A <p> és <strong> elemek megkeresése a DOM-fában
```

```

const p = document.getElementById("my-paragraph"); // külső elem
const strong = document.getElementById("my-strong"); // belső elem

// Kattintás eseményre vonatkozó eseménykezelők hozzárendelése az elemekhez
p.addEventListener("click", capture, true); // capturing
strong.addEventListener("click", capture, true); // capturing
p.addEventListener("click", bubble, false); // bubbling
strong.addEventListener("click", bubble, false); // bubbling
</script>
</body>

```

Az eseménykezelők paraméterként átadhatnak egy objektumot, aminek a tulajdonságai információkat adnak az esemény bekövetkezésének körülményeiről. Az interneten bőséges információ áll rendelkezésre a lehetséges paraméterekről. Az alábbi program kiírja, hogy melyik egérgombbal kattintottunk az oldalon lévő gombra.

```

<button onmousedown="egerGomb(event);">Kattints az egér
gombjaival!</button>
<script>
  function egerGomb(event) {
    console.log(`Az egérgomb sorszáma: ${event.button}`);
  }
</script>

```

Néhány elérhető tulajdonság a teljesség igénye nélkül:

- `event.button`: A lenyomott egérgomb. Bal gomb értéke 1, a jobb gomb értéke 2.
- `event.clientX`: Az esemény bekövetkezésének X koordinátája (oszlopa) képpontban.
- `event.clientY`: Az esemény bekövetkezésének Y koordinátája (sora) képpontban.
- `event.altKey`: Ez jelzi, hogy az Alt billentyűt az esemény során lenyomták-e.
- `event.ctrlKey`: Ez jelzi, hogy a Ctrl billentyűt az esemény során lenyomták-e.
- `event.shiftKey`: Ez jelzi, hogy a Shift billentyűt az esemény során lenyomták-e.
- `event.keyCode`: A lenyomott billentyű kódja (Unicode formátumban).
- `event.srcElement`: Az objektum, ahol az esemény bekövetkezett.

17.3 Feladatok

1. Készíts egy input mezőt, és használj eseménykezelőt annak figyelésére, ha a felhasználó beír valamit. Logold ki az írt szöveget a konzolra.
2. Készíts egy képet a weboldaladra, és alkalmazz egy eseménykezelőt, amely megváltoztatja a kép forrását, ha rákattintasz.
3. Hozz létre egy listát és egy gombot. Írj egy eseménykezelőt, amely hozzáad egy új elemet a listához, amikor a gombra kattintasz.
4. Készíts egy egyszerű számlálót egy gomb és egy szám megjelenítésére. Az eseménykezelő növelje a szám értékét minden kattintásnál.
5. Alkalmazz egy eseménykezelőt egy `mouseover` eseményre egy HTML elemen, amely megváltoztatja az elem háttérszínét.
6. Készíts egy formot, és írd meg eseménykezelőt, amely leállítja a form elküldését, majd logolja az űrlap adatait a konzolra.
7. Hozz létre egy egyszerű számológépet gombokkal és egy megjelenítővel. Írj eseménykezelőket a gombokhoz, hogy elvégezzenek alapműveleteket (pl.: összeadás, kivonás).
8. Készíts egy interaktív listát, ahol az eseménykezelő reagál az elemekre kattintva, és módosítja azok stílusát vagy tartalmát.
9. Alkalmazz egy eseménykezelőt az ablak `resize` eseményére, amely logolja a böngésző ablak méretét minden méretezésnél.
10. Hozz létre egy egyszerű játékot, ahol egy gombra kattintva egy véletlenszerű helyen jelenik meg egy másik gomb az oldalon.
11. Készíts egy időzítőt, amely egy gombra kattintva elindul, majd egy másik gombra kattintva leáll.
12. Hozz létre egy egyszerű slideshow-t képekkel, és írd meg eseménykezelőket egy előző és egy következő gombhoz.
13. Használj eseménykezelőt egy `keydown` eseményhez, hogy reagálj a billentyűleütésekre. Logolja ki a lenyomott billentyű kódját.
14. Készíts egy egyszerű drag-and-drop alkalmazást egy div elemmel, amit el lehet húzni a képernyőn.
15. Alkalmazz egy eseménykezelőt egy checkbox-hoz, és változtasd meg egy másik elem stílusát az állapotától függően.
16. Készíts egy űrlapot validációval, és használj eseménykezelőt a submit gombra kattintva az adatok ellenőrzésére.
17. Hozz létre egy időzített üzenetablakot, amely egy gombra kattintva megjelenik, majd néhány másodperc múlva eltűnik.
18. Készíts egy "névtelen szavazás" alkalmazást gombokkal, ahol az eseménykezelő rögzíti a szavazatokat, és megjeleníti az eredményt.
19. Használj eseménykezelőt a böngésző `scroll` eseményére, és változtasd meg egy elem stílusát az oldal görgetése során.

18 DOM manipuláció

18.1 A HTML DOM

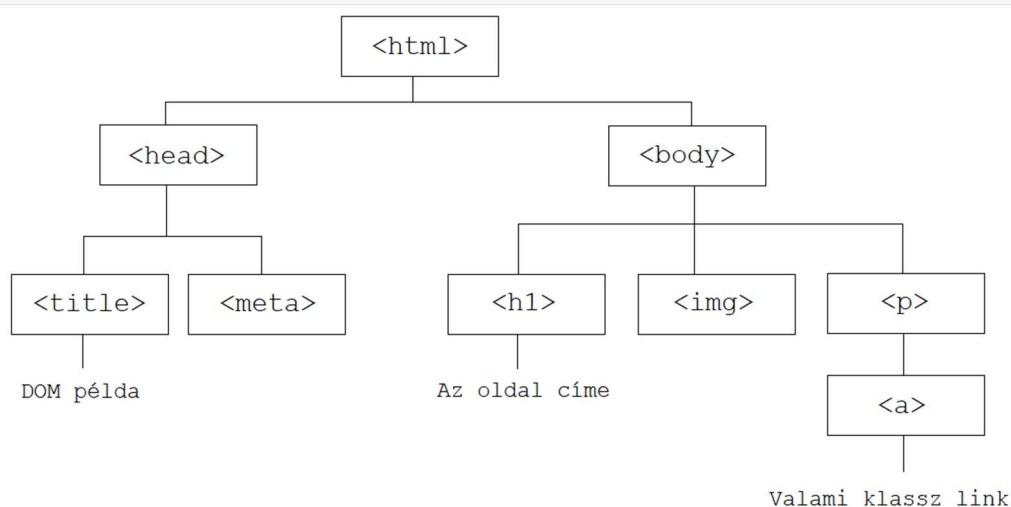
A webfejlesztésben a HTML nyelvet használjuk weboldalak létrehozására. Ennek a nyelvnek a segítségével mondhatjuk meg, hogy mi az, amit egy weboldalon látni szeretnénk (pl. szövegek, képek, táblázatok, űrlapok, multimédia stb.). Emellett a HTML lehetőséget biztosít a weboldalon megjelenő tartalom strukturálására is, különféle szakaszok, tartalmi egységek kialakításával.

A HTML dokumentumok úgy épülnek fel, hogy HTML objektumokat (úgynevezett tageket) ágyazunk egymásba. Ezek az objektumok egy hierarchikus fastruktúrát alkotnak a dokumentumban.

Amikor egy weboldal betöltődik, akkor a böngésző a weboldalon található HTML objektumokból elkészíti az úgynevezett dokumentum-objektum modellt, avagy röviden a DOM-ot. A DOM-fa (DOM tree) segítségével könnyen szemléltethetjük a weboldalon található HTML elemek hierarchikus viszonyait.

Példa: Egy egyszerű HTML kód és az ahhoz tartozó DOM-fa.

```
<!DOCTYPE html>
<html lang="hu">
  <head>
    <title>DOM példa</title>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <h1>Az oldal címe</h1>
    
    <p>
      <a href="https://www.youtube.com/watch?v=dQw4w9WgXcQ">Valami klassz
link</a>
    </p>
  </body>
</html>
```



DOM-fa

A `DOCTYPE` nem egy HTML tag, ezért a DOM-fában sem szerepel.

18.1.1 HTML elemek DOM-beli viszonyai

Ha egy HTML dokumentumban az A objektum (nem feltétlen közvetlenül) tartalmazza a B objektumot, akkor azt mondjuk, hogy az A objektum a B objektum őse, a B objektum pedig A-nak leszármazottja. Amennyiben ez a tartalmazás közvetlen, akkor A-t a B szülőjének, B-t pedig az A gyerekének nevezzük.

Néhány példa a fenti kódból és az ahhoz tartozó DOM-fából:

- A `<body>` objektum leszármazottjai a `<h1>`, ``, `<p>` és `<a>` objektumok, illetve a "Valami klassz link" és "Az oldal címe" szöveges csomópontok.
- A `<body>` objektum gyerekei a `<h1>`, `` és `<p>` objektumok. Másképp mondva: a `<h1>`, `` és `<p>` objektumok szülője a `<body>`.
- A `<body>` objektumnak az `<a>` objektum nem gyereke, csak leszármazottja, hiszen itt a tartalmazás nem közvetlen (van még a fában egy `<p>` elem is köztük).

Ha az A és B objektumok szülője megegyezik, akkor A és B egymás testvérei. Például a fenti kódban és az ahhoz tartozó DOM-fában a `<h1>`, `` és `<p>` elemek egymás testvérei, hiszen mindhárom elem szülője a `<body>`.

A DOM-fa tetején lévő, szülővel nem rendelkező elemet a fa gyökérelemének nevezzük. A teljes HTML DOM-ban a gyökérelem mindig a `<html>` objektum lesz (ugyanis ebbe ágyazunk be minden további HTML elemet).

18.1.2 A DOM gyakorlati jelentősége

Amikor a weboldalunk tartalmát CSS-ben formázzuk, akkor használhatunk olyan szelektorokat (kijelölőket) is, amelyek a DOM-beli viszonyaik alapján jelölnék ki HTML objektumokat. Néhány példa DOM-alapú CSS szelektorokra, a teljesség igénye nélkül:

- `div p`: kijelöl minden olyan `<p>`-t, amely egy `<div>` leszármazottja
- `div > p`: kijelöl minden olyan `<p>`-t, amely egy `<div>` gyereke
- `div ~ p`: kijelöl minden olyan `<p>`-t, amely egy `<div>` testvére és ezen `<div>` után szerepel
- `p:nth-child(n)`: kijelöl minden olyan `<p>`-t, amely a szülőjének `n`-edik gyereke
- `p:nth-of-type(n)`: kijelöl minden olyan `<p>`-t, amely a `<p>` típusú testvérei közül az `n`-edik.

A webes világban gyakran előfordul, hogy dinamikusan szeretnénk manipulálni a DOM-fát, miután már a weboldal betöltődött (pl. szeretnénk egy objektumot módosítani vagy törölni, esetleg egy új objektumot akarunk a fába beszúrni). Erre biztosítanak lehetőséget a JavaScript DOM-manipulációs műveletei.

A DOM tulajdonképpen nem más, mint egy objektumorientált reprezentációja a weboldalnak. A weboldalon szereplő elemek *Node*-ok (csomópontok) a DOM-fában, amelyek számos *property*-vel (adattaggal, tulajdonsággal) és *metódussal* rendelkeznek. Ezeket JavaScriptből egyszerűen el tudjuk érni.

A következő fejezetben megismerkedünk a JavaScript néhány fontosabb DOM-manipulációs lehetőségével. Fogjuk párszor használni a `document` objektumot, ami lényegében a böngésző által megnyitott HTML dokumentumot reprezentálja és hozzáférést biztosít a DOM-fához.

Oké, de mi értelme van ennek? Miért nem lehet csak simán a HTML-t átírogatni?

A hangsúly itt azon van, hogy azután szeretnénk a weboldal tartalmát dinamikusan módosítani, miután az oldal már betöltődött. Nézzünk néhány gyakorlati példát, amikor DOM-műveleteket használunk:

- Lekérdezzük adatokat egy szerverről, és meg szeretnénk jeleníteni azokat a weboldalunkon. Mivel a HTML oldal már betöltődött addigra, amire az adatok megérkeztek, ezért azokat úgy tudjuk megjeleníteni a weboldalon, hogy utólag szűrőgadjuk be őket a DOM-fába.
- Egy űrlap kitöltését követően kliensoldalon ellenőrizzük a beírt adatok helyességét (pl. helyes e-mail cím formátum, helyes születési dátum stb.). Ekkor egy eseménykezelővel figyeljük, hogy mikor nyomja meg a felhasználó az "Elküldés" gombot, majd a "klikk" esemény hatására DOM-műveletekkel lekérjük a beírt adatokat és ellenőrizzük őket.
- Szeretnénk elérni, hogy egy gombra kattintva a felhasználó válthasson világos és sötét téma között. Megint az a helyzet, hogy a weboldal már be van töltve, csupán annak a megjelenítését manipuláljuk dinamikusan, DOM-műveletek segítségével, amikor a felhasználó a gombra kattint.

Könnyen belátható, hogy a HTML önmagában nem elég robusztus ahhoz, hogy "utólag" manipuláljuk a weboldalaink szerkezetét. Ezért van szükségünk a DOM-ra és a JavaScript DOM-műveleteire.

18.2 JavaScript DOM-műveletek, egy példán keresztül

A fejezet hátralévő részében egy végletekig leegyszerűsített feladatlista alkalmazást fogunk elkészíteni. A weboldalon megjelennek a napi feladataink, amelyeket lehetőségünk van törölni, ha teljesítettük őket. Emellett új feladatot is bármikor létrehozhatunk.

Mivel csak a DOM-műveletek bemutatása a cél, ezért az alkalmazás eléggé kezdetleges lesz: a feladatokat nem mentjük el sehova, így az oldal frissítésekor a dinamikusan hozzáadott adatok elvesznek. A teljes kód az 1. mellékletben található.

18.2.1 Objektumok megkeresése a DOM-fában

Egy egyszerű feladattal fogunk indítani: keressük meg JavaScriptben az alábbi `<h1>`-es címsort a DOM-fában, és írassuk ki azt a konzolra!

```
<header>
  <h1 id="page-title" class="text-center">Feladataim</h1>
</header>
```

HTML objektumok DOM-fában történő megkeresésére többféle lehetőségünk is van:

- `document.getElementById(id)`: visszaadja az adott `id` értékkel rendelkező elemet (egyetlen elemet ad vissza, hiszen az `id` attribútum értéke szabályosan egyedi a HTML dokumentumon belül)
- `document.getElementsByTagName(tag)`: visszaadja az adott tagnévvel rendelkező elemeket (minden esetben egy kollekciót ad vissza, amit 0-tól kezdődően indexelünk)
- `document.getElementsByClassName(class)`: visszaadja az adott `class` értékkel rendelkező elemeket (minden esetben egy kollekciót ad vissza, amit 0-tól kezdődően indexelünk)
- `document.querySelector(s)`: visszaadja az `s` CSS szelektor által kijelölt legelső elemet

- `document.querySelectorAll(s)`: visszaadja az `s` CSS szelektor által kijelölt összes elemet (minden esetben egy kollekción ad vissza, amit 0-tól kezdődően indexelünk).

Példa: A felsoroltak közül bármelyik szelektorral megkereshetjük a fenti címsort. Az alábbi utasítások mindegyikének hatására a kérdéses címsor fog kiíródni a konzolra.

```
console.log(document.getElementById("page-title"));
console.log(document.getElementsByTagName("h1")[0]);
console.log(document.getElementsByClassName("text-center")[0]);
console.log(document.querySelector("header h1.text-center"));
console.log(document.querySelectorAll("header h1.text-center")[0]);
```

A `getElementsByTagName()`, a `getElementsByClassName()` és a `querySelectorAll()` egy kollekción ad vissza, melynek elemeit az indexük alapján érhetjük el!

18.2.2 Elemek beszúrása és módosítása

Ha egy új elemet szeretnénk a DOM-fába beszúrni, akkor a következő lépéseket kell követnünk:

1. A `document.createElement(tagname)` metódussal létrehozuk a beszúrandó HTML elemet.
2. Beállítjuk az újonnan létrehozott elem tartalmát, attribútumait és stílusát (opcionális lépés).
3. Beszúrjuk az elemet a DOM-fába a szülőobjektum `append()` vagy `appendChild()` metódusával.

Az `append()` és `appendChild()` metódusok mindketten arra szolgálnak, hogy egy DOM-beli objektumhoz gyerekobjektumot fűzzünk hozzá. Két fontos különbség a két metódus között:

- Az `append()` egyszerre több gyereket is hozzáfűzhet egy objektumhoz, míg az `appendChild()` csak egyet.
- Az `append()` Node-okat és stringeket egyaránt hozzáfűzhet egy objektumhoz gyerekként, míg az `appendChild()` kizárólag Node-okkal működik.

Az `append()` és `appendChild()` metódusok mindig a szülőelem legutolsó gyerekeként szúrják be az új objektumot a DOM-fába. Ha a szülő egy tetszőleges indexű gyerekeként szeretnénk beszúrni az elemet a fába, akkor használjuk a szülő `insertBefore()` metódusát.

A metódus két paramétert vár: rendre a beszúrandó objektumot, és a szülőelem azon gyerekeit, ami elé be fogjuk szúrni az új elemet. A szülőelemnek felhasználhatjuk a `children` property-jét, ami visszaadja az elem összes gyerekeit egy indexelhető kollekciónban.

Nézzünk egy példát! Szúrjunk be az alábbi weboldalra egy "Második bekezdés" feliratú `<p>` objektumot, a "Harmadik bekezdés" feliratú `<p>` elem elé!

```
<div id="parent">
  <p>Első bekezdés</p>
  <p>Harmadik bekezdés</p>
</div>
<script>
  const parent = document.getElementById("parent"); // a szülőobjektum
  const newParagraph = document.createElement("p"); // elem létrehozása
  newParagraph.innerText = "Második bekezdés";
  // Az új elem beszúrása a szülőobjektum második (1. indexű) gyereke elé
  parent.insertBefore(newParagraph, parent.children[1]);
</script>
```

Feladat: Tegyük működőképpé az "új feladat hozzáadása" funkciót a példaprojektben!

A feladat hozzáadását végző űrlap HTML kódja a következő:

```
<form>
  <input type="text" id="task-text" class="form-input"/>
  <button type="button" class="add-btn"
onclick="addTask()" >Hozzáadás</button>
</form>
```

Látható, hogy a "Hozzáadás" gombhoz hozzárendeltük az `onclick` attribútummal az `addTask()` eseménykezelő függvényt. Ez a függvény fog meghívódni, amikor a felhasználó a gombra kattint, így ennek a törzsét kell megírunk. Azt szeretnénk, hogy a függvény kérdezze le az űrlapmezőbe írt szöveget és szűrje be azt és egy "Törlés" gombot a weboldalon található táblázat egy új sorába. Valahogy így:

```
<table>
  <!-- A táblázat fejléce... -->
  <tbody>
    <!-- Néhány korábbi feladat... -->
    <tr>
      <td>Az újonnan beszúrt feladat szövege...</td>
      <td><button type="button" class="delete-btn"
onclick="deleteTask(this)" >X</button></td>
    </tr>
  </tbody>
</table>
```

A feladat megoldásának lépései

1. Keressük meg a `<tbody>` objektumot, hiszen ennek a gyerekeként fogjuk az új feladatot beszúrni!
2. Hozzunk létre a beszúrni kívánt feladatnak egy új sort a táblázatban (`<tr>`)! A sorban helyezünk el két táblázatcellát (`<td>`)!
3. Az első táblázatcellába írjuk bele a feladat szövegét, vagyis az űrlapon található beviteli mezőbe írt értéket!
4. A második táblázatcellában helyezünk el egy "Törlés" gombot, a fenti kódban található mintának megfelelően!
5. Szűrjük be a két `<td>` objektumot a `<tr>` gyerekeiként a DOM-fába! A `<tr>` objektum pedig legyen a `<tbody>` gyereke a weboldalon!

Keressük meg a `<tbody>` objektumot!

```
const tbody = document.getElementsByTagName("tbody")[0];
```

Hozzunk létre egy táblázatsort és két táblázatcellát! A cellákat a sor gyerekeiként, a sort pedig a `<tbody>` gyerekeként szűrjük be a DOM-fába!

```
const tbody = document.getElementsByTagName("tbody")[0];

const row = document.createElement("tr");
const column1 = document.createElement("td");
// Itt majd kialakítjuk az első cella tartalmát...
const column2 = document.createElement("td");
// Itt majd kialakítjuk a második cella tartalmát...

row.append(column1, column2);
tbody.append(row);
```

Írjuk bele az első táblázatcellába az `id="task-text"` attribútummal rendelkező beviteli mezőbe írt szöveget!

- Egy beviteli mezőbe írt érték lekérdezése a mező `value` property-jével történik.
- Egy HTML objektum szöveges tartalmának beállítása az `innerText` vagy `innerHTML` property-vel lehetséges. A különbség a két property között, hogy az `innerHTML` értékeként megadott szöveg HTML-ként lesz értelmezve (ezért itt használhatók a szokásos HTML tagek), míg az `innerText` értéke minden esetben egyszerű szöveggént jelenik meg (nem lesz HTML-ként értelmezve).

```
// ...
const column1 = document.createElement("td");
column1.innerText = document.getElementById("task-text").value;
// ...
```

Már csak a második táblázatcella tartalmát kell kialakítanunk. Ebben egy törlés gombot fogunk elhelyezni, tehát a `document.createElement()` módszerrel létrehozunk egy új `<button>` objektumot, amit a második táblázatcella gyerekeként szúrunk be a DOM-fába. A gomb szöveges tartalma legyen `x`!

Ahhoz, hogy a létrehozott gomb megfelelően működjön, hozzáadunk néhány attribútum-érték párt. Egy HTML objektum attribútumainak beállítása a `setAttribute()` módszerrel történik. Ennek első paramétereként megadjuk a beállítani kívánt attribútum nevét, második paraméterként pedig az attribútum értékét.

- A `setAttribute()` segítségével adjunk a gombnak egy `type="button"` attribútum-érték párt!
- A `setAttribute()` segítségével adjunk a gombnak egy `onclick="deleteTask(this)"` attribútum-érték párt, amivel hozzárendeljük a gombhoz a `deleteTask()` eseménykezelő függvényt!

Rendeljük hozzá a gombhoz a `class="delete-btn"` attribútumot, hogy ugyanúgy legyen formázva, mint a többi törlés gomb! Ehhez használjuk az elem `classList` property-jét, amivel lekérhetjük az összes olyan class nevét, amivel az elem rendelkezik. A `classList`-nek az `add()` módszerével hozzáadjuk a gombhoz a `delete-btn` class-értéket.

```
// ...
const column2 = document.createElement("td");
const deleteBtn = document.createElement("button"); // gomb létrehozása
deleteBtn.innerText = "X"; // gomb szöveges
// tartalmának beállítása
deleteBtn.setAttribute("type", "button"); // gomb
// attribútumainak beállítása
deleteBtn.setAttribute("onclick", "deleteTask(this)");
deleteBtn.classList.add("delete-btn"); // class="delete-btn" attribútum
// hozzáadása a gombhoz
column2.append(deleteBtn); // gomb beszúrása a DOM-fába a 2.
// táblázatcella gyerekeként
// ...
```

Tehát az `addTask()` függvény végleges verziója a következőképpen néz ki:

```
function addTask() {
  const tbody = document.getElementsByTagName("tbody")[0];
  const row = document.createElement("tr");

  const column1 = document.createElement("td");
  column1.innerText = document.getElementById("task-text").value;

  const column2 = document.createElement("td");
  const deleteBtn = document.createElement("button");
  deleteBtn.innerText = "X";
  deleteBtn.setAttribute("type", "button");
  deleteBtn.setAttribute("onclick", "deleteTask(this)");
  deleteBtn.classList.add("delete-btn");
  column2.append(deleteBtn);

  row.append(column1, column2);
  tbody.append(row);
  document.getElementById("task-text").value = ""; //űrlapmezőkiürítése
}
```

A `classList` egyéb metódusai a class-hozzáadásra szolgáló `add()`-on kívül:

- `remove(c)`: a `c` class eltávolítása az elemről
- `contains(c)`: szerepel-e a `c` class az elemen
- `toggle(c)`: ha a `c` class még nem szerepel az elemen, akkor hozzáadja azt az elemhez; ha pedig már szerepel, akkor eltávolítja azt az elemről.

18.2.3 Objektumok törlése

18.2.3.1 Egy adott gyerekobjektum törlése

Ahhoz, hogy egy objektumot kitöröljünk a DOM-fából, szükségünk lesz a törlendő objektumra és annak szülőjére. A törléshez a szülő `removeChild()` metódusának paramétereként adjuk meg a törlendő objektumot.

Feladat: Tegyük működőképpessé a táblázat soraiban megjelenő "Feladat törlése" gombokat!

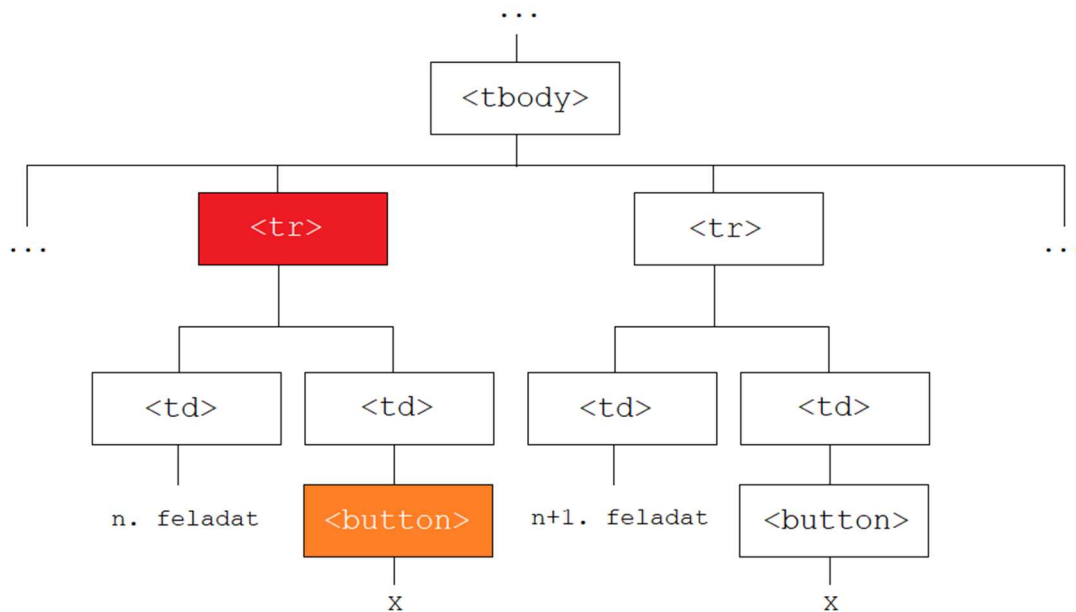
Minden ilyen, feladat törlésére szolgáló gomb forráskódja a következő:

```
<button type="button" class="delete-btn"
onclick="deleteTask(this)">X</button>
```

A gombra kattintva tehát a `deleteTask()` eseménykezelő függvény hívódik meg, így ennek a törzsét kell megírunk. Azt szeretnénk, hogy a gombra kattintva töröljük ki a táblázatból azt a sort, amihez a gomb tartozik. A függvény paraméterben megkapja az aktuális objektumot (`this`), azaz a gombot, amire kattintottunk.

A törlendő sorhoz tartozó gomb tehát a függvény paramétereként adott. Ahhoz, hogy ebből megkapjuk a törlendő sort, végig kell gondolnunk a gomb és az őt tartalmazó táblázatsor viszonyát. A törlés gombok szülője egy `<td>` (cella), amelynek szülője lesz a törlendő `<tr>` (sor). Tehát két szülővel kell "feljebb lépünk" a DOM-ban a megnyomott gombhoz képest. Kelleni fog még a törlendő sor szülője is, ami a `<tbody>` objektum lesz.

Mindez egy ábrán szemléltetve:



DOM-fa részlet: A törlendő sor a megnyomott gomb "nagyszülője" a fában

A feladat megoldásának lépései:

1. Keressük meg a `<tbody>` objektumot, hiszen ennek az egyik gyerekét (egy táblázatsort) fogjuk kitörölni!
2. Keressük meg a megnyomott gomb "nagyszülőjét" (azaz a szülőjének a szülőjét), hiszen ez lesz a törlendő sor!
3. A `<tbody>` objektum `removeChild()` metódusával töröljük a megtalált sort a `<tbody>` gyerekei közül!

A DOM-beli elemek `parentNode` property-jével lekérdezhetjük az adott elem szülőjét. Ahhoz, hogy megkapjuk a megnyomott gomb "nagyszülőjét" (a törlendő sort), kétszer egymás után alkalmazzuk a property-t.

A `deleteTask()` függvény végleges verziója a következőképpen néz ki:

```
function deleteTask(btn) {  
  const tbody = document.getElementsByTagName("tbody")[0];  
  //<tbody> megkeresése  
  const row = btn.parentNode.parentNode;  
  // a törlendő sor a megnyomott gomb "nagyszülője" lesz  
  tbody.removeChild(row);  
  // a sor kitörlése a <tbody> gyerekei közül  
}
```

18.2.3.2 Összes gyerekobjektum törlése

Feladat: Tegyük működőképpé a weboldalon található "Összes feladat törlése" gombot!

```
<button type="button" id="delete-all-tasks-btn" class="delete-btn"  
onclick="deleteAllTasks()">  
  Összes feladat törlése  
</button>
```

A gombra kattintva tehát a `deleteAllTasks()` eseménykezelő függvény hívódik meg, így ennek a törzsét kell megírunk. Azt szeretnénk, hogy a függvény törölje ki az összes feladatot a DOM-fából. A feladatok továbbra is a `<tbody>` gyerekeiként, táblázatsorok formájában vannak jelen a DOM-ban.

JavaScriptben nincs olyan explicit DOM-metódusunk, amely egy objektum összes gyerekeit kitörölné, ezért egy kicsit másképp oldjuk meg a dolgokat:

- A szülőobjektumnak (aminek a gyerekobjektumait törölni akarjuk) a `hasChildNodes()` metódusával lekérdezzük, hogy vannak-e gyerekei.
- Amíg vannak a szülőnek gyerekei, addig a `removeChild()` metódussal mindig kitörölünk egy gyereket.

A feladat megoldásának lépései

1. Keressük meg a `<tbody>` objektumot, hiszen ennek az összes gyereket (a feladatokat tartalmazó táblázatsorokat) fogjuk kitörölni!
2. Amíg van a `<tbody>`-nak gyereke, addig mindig töröljük ki egy tetszőleges gyereket!

Teljesen mindegy, hogy mikor melyik gyereket töröljük ki, hiszen végül minden gyereket ki fogunk törölni. A példánkban én mindig a `<tbody>` legelső gyereket törlöm, amit a szülő `firstChild` property-jével érhetünk el.

Tehát a `deleteAllTasks()` függvény végleges verziója a következőképpen néz ki:

```
function deleteAllTasks() {
  const tbody = document.getElementsByTagName("tbody")[0];
  // <tbody> megkeresése

  while (tbody.hasChildNodes()) { // <tbody> összes gyerekének törlése
    tbody.removeChild(tbody.firstChild);
  }
}
```

18.3 Stíluselemek megváltoztatása

A HTML DOM lehetőséget nyújt a stíluselemek megváltoztatására is. Miután megkerestük a HTML elemet, aminek a stílusát szeretnénk megváltoztatni, `style.stílustulajdonság`-nak adunk új értéket.

Fejlesszük tovább a programunkat, úgy hogy a feladatok táblázat csak akkor jelenjen meg, ha van benne adat. Ehhez az eltüntetni kívánt részt egy `div`-ben helyezzük el:

```
<div id="task-container" style="display: none;">

</div>
```

A `style="display: none;"` inline stílusbejegyzés miatt a `div` tartalma nem jelenik meg az oldalon.

Készítsünk két függvényt, ami ennek a stílusbejegyzésnek a tartalmát módosítja:

```
function hideTaskContainer() {
  document.getElementById("task-container").style.display = "none";
}
```



```
function showTaskContainer() {
    document.getElementById("task-container").style.display = "block";
}
```

Az `addTask()` függvény végén hívjuk meg a `showTaskContainer()` függvényt, mert amikor hozzáadunk egy új feladatot a táblázathoz, akkor annak meg kell jelennie. Ugyanígy a `deleteAllTasks()` függvény végén a `hideTaskContainer()` függvénnyel eltüntetjük az egész szakaszt, mert nem marad megjelenítendő adat.

A `deleteTask(btn)` függvény végén ellenőriznünk kell, hogy van-e még node a `tbody` objektumban. Ha nincs, akkor meghívjuk a `hideTaskContainer()` metódust.

```
if (!tbody.hasChildNodes()) {
    hideTaskContainer();
}
```

Ezzel a példaprogramunk elkészült. Most nézzünk pár példát, ami bemutatja, hogyan lehet a stílusbejegyzéseket módosítani a html DOM-ban.

Az alábbi példában a címsor háttérszínét változtatjuk meg.

```
<h1 id="cimsor">Háttérszín változtatás</h1>
<button type="button"
onclick="document.getElementById('cimsor').style.backgroundColor = 'red'">
Piros háttér</button>
```

A másik példában a gomb megnyomásakor eltűnik a címsor.

```
<h1 id="cimsor">El fogok tűnni</h1>
<button type="button"
onclick="document.getElementById('cimsor').style.visibility = 'hidden'">
Címsor eltüntetése</button>
```

Az oldal tulajdonságait is meg tudjuk változtatni. Például a háttérszínt:

```
<button type="button"
onclick="document.body.style.backgroundColor = 'red'">
Az oldal háttere legyen piros</button>
```

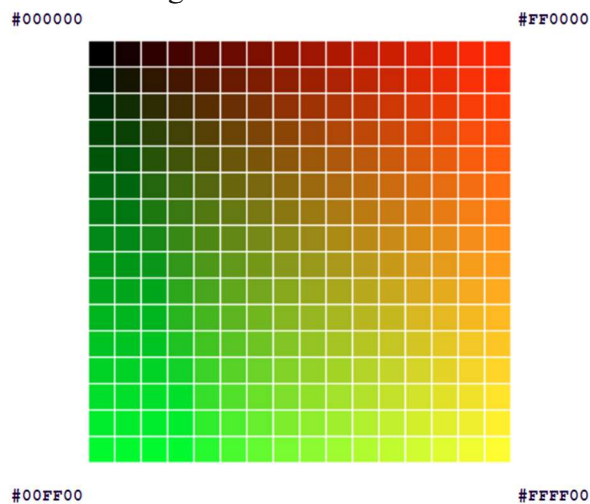
18.4 Feladatok

1. Elem kiválasztása: Válaszd ki a HTML oldal egy elemét és logold ki a konzolra annak tartalmát.
2. Elem módosítása: Módosítsd meg egy elem tartalmát JavaScript segítségével.
3. Elem hozzáadása: Adj hozzá egy új elemet a HTML oldalhoz JavaScript segítségével.
4. Elem eltávolítása: Távolítsd el egy elemet a HTML oldalról JavaScript használatával.
5. Attribútum módosítása: Módosítsd meg egy elem attribútumának értékét JavaScript segítségével.
6. Szülő elem kiválasztása: Válaszd ki egy elem szülőjét és logold ki annak tartalmát a konzolra.
7. Gyermekelemek kiválasztása: Válaszd ki egy elem gyermekeit és logold ki azok tartalmát a konzolra.
8. Elem stílusának módosítása: Módosítsd meg egy elem stílusát (pl. háttérszín, betűméret) JavaScript használatával.

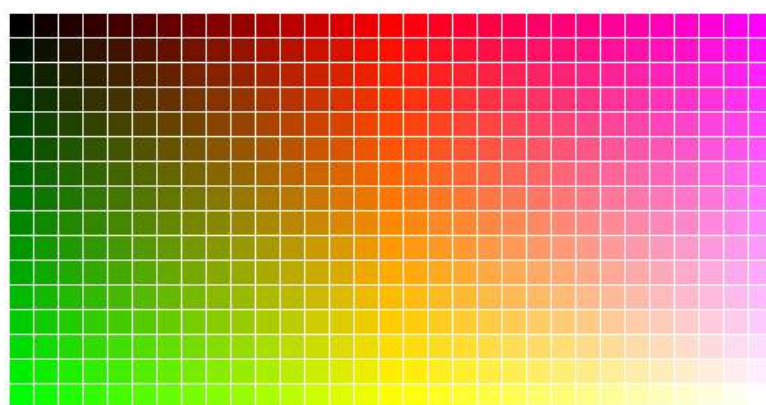
9. Készítsünk böngészőben megjelenő szorzótáblát az alábbi ábra mintájára! Ha a táblázat egy belső cellájára ráhúzzuk az egeret, akkor felvillanó feliratként jelenjen meg, mely két szám szorzatának eredménye látható a cellában.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160
9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180
10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
11	22	33	44	55	66	77	88	99	110	121	132	143	154	165	176	187	198	209	220
12	24	36	48	60	72	84	96	108	120	132	144	156	168	180	192	204	216	228	240
13	26	39	52	65	78	91	104	117	130	143	156	169	182	195	208	221	234	247	260
14	28	42	56	70	84	98	112	126	140	154	168	182	196	210	224	238	252	266	280
15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320
17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340
18	36	54	72	90	108	126	144	162	180	198	216	234	252	270	288	306	324	342	360
19	38	57	76	95	114	133	152	171	190	209	228	247	266	285	304	323	342	361	380
20	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400

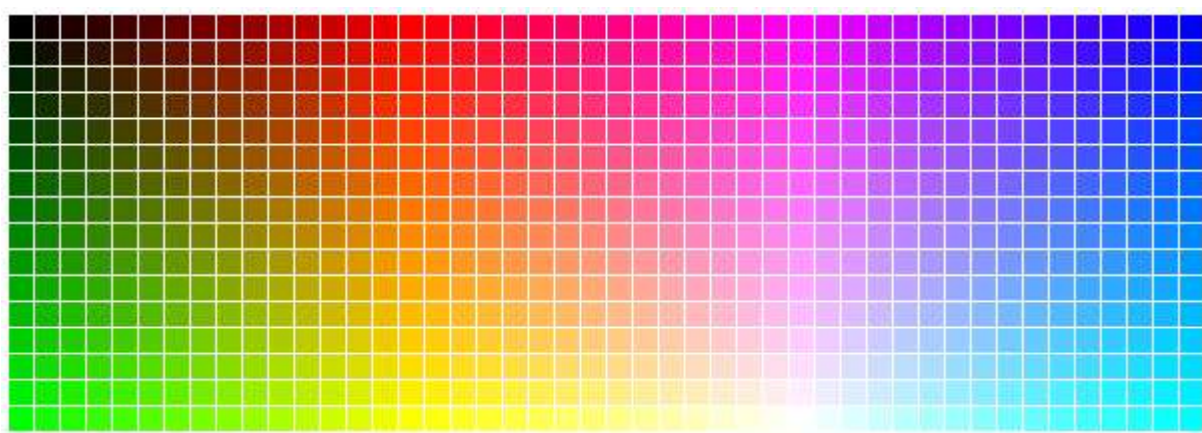
10. Készítsen programot, mely az alábbi színtáblázatot jeleníti meg böngészőben, a sarkokon jelölt színkódoknak megfelelően!



Egészítse ki a kezdeti ábrát jobbra: a folytatásban a jobb szélén fent a bíbor (#FF00FF), lent pedig a fehér (#FFFFFF) színekig alakítsa a négyzeteket.



Mivel a színek közül még hiányzik a kék (**#0000FF**), és a cián (**#00FFFF**), ezért bővítse tovább a táblázatot ezekkel is.



19 Űrlapok használata

A JavaScript egyik leggyakoribb felhasználási területe az űrlapok beviteli mezőinek ellenőrzése. Ha a mezők ellenőrzése csak szerveroldalon történne, minden egyes hiba észrevételekor az űrlapot vissza kéne küldeni, ezzel feleslegesen terhelve a hálózatot, és fogyasztva az időt.

Az űrlap adatit legegyszerűbben a `name` attributum segítségével érhetjük el.

```
<form action="" name="urlap" onsubmit="ellenor();" >
  <p>Nev: <input type="text" name="nev"></p>
  <input type="submit" value="Elküld">
</form>
```

A mezők helyességének ellenőrzéséhez reguláris kifejezéseket is használhatunk. Az alábbi függvényeket a fejrészben definiáljuk, majd a Submit gomb lenyomásakor (onsubmit) meghívjuk, így az űrlap elküldése előtt még ellenőrzi a mezők tartalmát.

```
<form action="" name="urlap" onsubmit="ellenor();" >
  <p>Telefon: <input type="text" name="tel"></p>
  <p>E-mail: <input type="text" name="mail"></p>
  <input type="submit" value="Elküld">
</form>
```

A telefonszám ellenőrzése a következőképpen zajlik: először kiszedjük az elválasztójelek mindegyikét (ezért kell a `'/'` kapcsoló), majd ellenőrizzük a `test` függvénnyel, hogy csak számjegyek szerepelnek-e benne.

```
function telefonEllenor(strng) {
  error = "";
  let newstrng = strng.replace( /[\\(\\)\\.\\-\\ ]/g , '' );
  let filter = /^\\d+$/;
  if(!filter.test(newstrng)) {
    error = "Kérem adjon meg érvényes telefonszámot!\\n";
  }
  return error;
}
```

Az következő függvényben szereplő minta jelentése: 1 vagy több bármilyen karakter, majd `@` jel, aztán megint 1 vagy több bármilyen karakter, egy pont és utána 2-től 6 hosszúságú karakterlánc.

```
function emailEllenor(strng) {
  error = "";
  let filter = /^.+@.+\\.\\. {2,6}$/;
  if(!filter.test(strng)) {
    error = "Kérem adjon meg érvényes e-mail címet!\\n";
  }
  return error;
}
```

Az `onsubmit` eseményre az `ellenor()` függvényt hívjuk meg, ami `allert` ablakba írja az esetleges hibaüzeneteket.

```
function ellenor() {
    let tel = document.forms['urlap']['tel'].value;
    let mail = document.forms['urlap']['mail'].value;
    if (telefonEllenor(tel)) {
        alert(error);
    } else if (emailEllenor(mail)) {
        alert(error);
    }
}
```

19.1 Feladatok

1. Készíts egy űrlapot egy egyszerű szöveges beviteli mezővel. Adj hozzá egy gombot az űrlaphoz, amely kiírja a beírt szöveget egy üzenetként az oldalon.
2. Hozz létre egy űrlapot, amely tartalmaz egy szám beviteli mezőt. Adj hozzá egy gombot az űrlaphoz, amely kiírja a beírt számot a képernyőn.
3. Készíts egy űrlapot egy legördülő listával. Írj JavaScript kódot, amely kiírja a kiválasztott értéket, amikor a felhasználó választott a listából.
4. Készíts egy űrlapot egy jelölőnégyzettel (checkbox), ami kiírja, hogy melyik négyzetek vannak kiválasztva.
5. Hozz létre egy űrlapot, amely tartalmaz egy többsoros szöveges beviteli mezőt (textarea). Írj egy JavaScript függvényt, amely ellenőrzi, hogy a beírt szöveg eléri-e a megadott hosszúságot, és kiírja ennek megfelelő üzenetet.
6. Készíts egy űrlapot egy radio gomb csoporttal. Írj egy JavaScript függvényt, amely kiírja a kiválasztott rádiógomb értékét.
7. Hozz létre egy űrlapot egy dátum beviteli mezővel. Írj JavaScript kódot, amely ellenőrzi, hogy a választott dátum megfelelő-e (például, nem lehet jövőbeli dátum).
8. Készíts egy űrlapot egy számkiválasztóval (number input). Írj egy JavaScript függvényt, amely duplázza meg a beírt számot, amikor a felhasználó elküldi az űrlapot.
9. Hozz létre egy űrlapot egy e-mail beviteli mezővel. Írj JavaScript kódot, amely ellenőrzi, hogy az e-mail cím megfelel-e a helyes formátumnak.
10. Készíts egy űrlapot egy jelszó beviteli mezővel. Írj egy JavaScript függvényt, amely ellenőrzi a jelszó erősségét (például: minimum hosszúság, kis- és nagybetűk, számok).
11. Készíts egy összetett űrlapot, amely többféle típusú beviteli mezőt tartalmaz, és írd JavaScript kódot, amely ellenőrzi az összes mező érvényességét.
12. Írj programot, ami egy űrlapon bekér két számot, gombnyomással lehessen kiválasztani a négy alapművelet egyikét és írjuk ki az eredményt.