JavaScript

Ezt a tananyagot innen lehet letölteni: http://szilagyidonat.hu/tananyag.html

Szerző: Szilágyi Donát

E-mail: donat_szilagyi@hotmail.com

1 JavaScript használata HTML oldalon

HTML fájlba ágyazott JavaScript

Készítsünk egy html kiterjesztésű fájlt. (Total Commander: shift +F4) A neve lehet például 1.html, a lényeg, hogy **.html** legyen a vége. Ezt írjuk bele:

```
<script>
document.write('Hello');
</script>
```

Ha dupla klikkel futtatjuk a fájlt, a böngészőben a Hello szöveg fog megjelenni.

Ami a **<script>** és a **</script>** elem (másnéven: tag) között van az a JavaScript. Élé és mögé persze tetszőleges egyéb HTML elemeket is tehetünk.

Ha kevesebbet akarunk gépelni, a **Ctrl** és a **C** gombbal tudunk egy kijelölt szöveget lemásolni, és a **Ctrl** és **V** gombbal beilleszteni, amit kimásoltunk.

Szöveg kiírása

Szöveget az előbb látott **document.write** segítségével írhatunk ki. Ez a HTML dokumentumban fog megjelenni.

Ezenkívül használhatjuk a **window.alert**-et is:

```
<script>
window.alert('Hello2');
</script>
```

A kiírt szöveget szimpla **'** vagy dupla **"** macskakörömbe kell tenni. Ha csak számot írunk ki, azt nem kell macskakörömbe tenni.

A JavaScript parancsokat pontosvesszővel szoktuk lezárni. Egy sorba több parancsot is írhatunk, de inkább külön sorba szokás írni őket.

HTML-ben a kis- és nagybetű nem számít, JavaScript-ben viszont számít (angolul: case-sensitive).

Egy harmadik módszer a **console.log**. Ezzel a böngésző konzoljára írhatunk olyan szöveget. Ide olyan szöveget szoktunk írni, ami csak a program íróját érdekli. Pl. 'itt valami nem működik'.

A konzolt az alábbi módon lehet bekapcsolni:

- **Firefox**: jobb felső sarok 3 vízszintes vonalra klikk → Web Developer → Web Console
- **Google Chrome**: jobb felső sarok 3 pontra klikk → More tools → Developer tools → alul: Console fül (fül = tab)

- **Internet Explorer**: jobb felső sarok fogaskerékre klikk → F12 Developer Tools → alul: Console fül

```
<script>
console.log('Hello3');
</script>
```

Ha a konzolon valami ilyesmi szöveg is megjelenik: The character encoding of the HTML document was not declared. Azt nem kell komolyan venni.

A konzol azért nagyon fontos, mert ha nem működik a programunk, a böngésző ide írja ki a hibák okát. Szóval programozás közben sokszor be szoktuk kapcsolni.

Bármelyik böngészőt (= browser) használjuk, az **F5** gomb segítségével újratölthetjük a HTML oldalt, és ekkor a rajta lévő JavaScript kód is újra le fog futni.

Továbbá szeretnék mindenkit arra biztatni, hogy a példaprogramokat próbálja ki, és kísérletezésképpen próbáljon apró módosításokat végezni benne. Ez még annál is fontosabb, minthogy eljussunk az egész tananyag végéig.

KÜLÖN JAVASCRIPT FÁJL

A hosszabb JavaScript programokat külön **.js** kiterjesztésű fájlokba is tehetjük:

x.html

```
<script src="x.js"></script>
```

x.js

```
document.write('Hello JavaScript!');
```

A programot az **x.html**-en történő dupla klikkel indíthatjuk. Ha az **x.js**-t ugyanabba a könyvtárba tettük, akkor az **x.html** meg fogja találni.

ESEMÉNYKEZELÉS

A HTML fájlba nemcsak a <SCRIPT> elem segítségével írhatunk bele JavaScript parancsokat. Használhatjuk a HTML elemek on-nal kezdődő paramétereit is: onclick, ondblclick, onload, onchange, onmouseenter, onkeydown. Ezekből elég sok van még, úgy hívják őket, hogy HTML DOM eseménykezelők, vagy angolul HTML DOM event handler.

Példaként írjuk ezt egy HTML fájlba:

```
<div onclick="window.alert('Hello');">Klikkelj ide!</div>
```

És klikkeljünk a feliratra.

2 JAVASCRIPT ALAPOK

KOMMENT

Kommenteket akkor használunk, ha a programból néhány sort ideiglenesen ki akarunk kapcsolni, vagy ha valamilyen megjegyzést akarunk írni a programba.

Egysoros kommentet a **//** segítségével csinálhatunk. A **//** -től jobbra lévő szöveget vagy programrészt a böngésző nem fogja futtatni.

Ha többsoros kommentet akarunk akkor azt **/*** és ***/** közé kell tennünk.

A profi programozók nagyon sok kommentet használnak, abban magyarázzák el mit csinálnak a program részei.

```
<script>
document.write('hello');
// document.write('ez inkább ne fusson le');
/*
ide még kellene valami
*/
</script>
```

Az eredmény: hello.

VÁLTOZÓK, SZÁM, SZÖVEG

A változókat a **var** kulcsszóval adhatjuk meg.

```
var szam1 = 3;
```

Van egy **nev**ük (szam1) és egy **érték**ük (3). A név és érték angolul: **name**, **value**.

Az értékük lehet **szöveg** (angolul: **string**), **szám** (angolul: **number**), vagy néhány más dolog is.

Két számot összeadhatunk, kivonhatunk, összeszorozhatunk és eloszthatunk egymással az alábbi műveleti jelekkel: **+ - * /**. Használhatunk zárójeleket is.

```
<script>
var szam1 = 3;
var szam2 = 2;
```

```
document.write((szam1 + szam2) * 4);
</script>
```

Az eredmény 20 lesz, mert (3 + 2) * 4 = 20.

Szöveget csak összeadni tudunk más szöveggel vagy számmal. Ilyenkor az összeadott szövegekből és számokból egy új szöveg lesz, amiben egymás után szerepelnek, amiket összeadtunk.

```
<meta charset="utf-8"/>
<script>
var kor = 10;
var nev = 'Cinc';
document.write(nev + ' <b>' + kor + '</b> éves.');
</script>
```

Az eredmény: Cinc **10** éves.

A **meta** elemre csak azért van szükség, hogy tudja értelmezni a magyar é betűt. Azt jelenti, hogy **UTF-8** kódolásban értelmezi a magyar ékezetes betűket. Győződjünk meg róla, hogy a **Notepad++** is UTF-8 kódolásra van-e állítva: **Encoding** → **Encode in UTF-8**. Ha nem arra van állítva, akkor állítsuk arra.

A példában nem csak változókat, hanem macskakörmök közti fix szövegeket is összeadtunk. A változókban vagy a fix szövegekben lehetnek akár HTML elemek is, mint például itt a **<b>**.

A **var** kulcsszót csak az első alkalommal kell használni. Ha egy változónak új értéket adunk, akkor már nem kell **var**.

```
<script>
var x = 'Hello';
x = 'Szia';
document.write(x);
</script>
```

Az eredmény Szia lesz.

A **window.prompt** feldob egy kis ablakot, amiben bekérdezhetjük egy változó értékét:

```
<meta charset="utf-8"/>
<script>
var nev = window.prompt('Mi a neved?');
document.write('Szia ' + nev + '!');
</script>
```

NÉHÁNY MŰVELET SZÖVEGGEL

A **.length** visszaadja a szöveg hosszát. Ebbe az üres helyek is beleszámítanak.

```
<script>
   var szoveg = 'Szia Mia!';
   document.write(szoveg.length);
</script>
```

Az eredmény: 9

A **.toLowerCase()** csupa kisbetűssé alakít egy szöveget, a **.toUpperCase()** pedig csupa nagybetűssé.

```
<script>
   var szoveg = 'Szia Mia!';
   document.write(szoveg.toLowerCase());
   document.write(szoveg.toUpperCase());
</script>
```

Az eredmény: szia mia!SZIA MIA!

Ha a két sorban akarjuk megjeleníteni, ki kell íratnunk egy **
** HTML elemet is.

```
<script>
   var szoveg = 'Szia Mia!';
   document.write(szoveg.toLowerCase() + '<br>');
   document.write(szoveg.toUpperCase());
</script>
```

Az **.indexOf()** segítségével megkereshetünk egy szövegdarabot egy másik szövegben.

```
<script>
   var szoveg = 'Szia Mia!';
   document.write(szoveg.indexOf('Szia') + '<br>');
   document.write(szoveg.indexOf('Mia') + '<br>');
   document.write(szoveg.indexOf('Hola'));
</script>
```

Az eredmény: 0 5 -1

A fenti példában a **szoveg.indexOf('Szia')** eredménye nulla, mert a 'Szia Mia!' -n belül a nulladik helyen van a 'Szia'. A számozás nullával kezdődik. A 'Mia' az 5. helyen van.

S	z	i	a		M	i	a	!
0	1	2	3	4	5	6	7	8

Ha az .indexOf() nem találja meg a szövegdarabot, akkor mínusz egyet ad vissza.

BOOLEAN

A változók értéke lehet szövegen és számon kívül **boolean** is. A boolean-nak kétféle értéke lehet: **true**, **false**. (true = igaz, false = hamis)

```
<script>
var x = false;
document.write(x);
var y = 10 > 5;
document.write(y);
</script>
```

Ez azt fogja kiírni, hogy falsetrue.

Ha külön sorba akarjuk írni, egy **
** -t is ki kell íratni. Ehhez nem kell új document.write, a meglévőhöz is hozzácsaphatjuk:

```
document.write(x + '<br>');
```

Ha boolean-t és szöveget adunk össze, abból is szöveg lesz.

Boolean eredményt kapunk, ha két számot összehasonlítunk egymással. Az alábbi műveleteket használhatjuk:

<	kisebb
<=	kisebb egyenlő
>	nagyobb
>=	nagyobb egyenlő
==	egyenlő
!=	nem egyenlő

Próbáljuk ki:

```
<script>
document.write(10 > 5);
document.write(10 <= 5);
document.write(10 == 5);
var husz = 20;
document.write(husz == 20);
</script>
```

Az eredmény truefalsefalsetrue lesz.

Szövegeket is összehasonlíthatunk egymással:

```
<script>
document.write('hello' == 'hello');
document.write('hello' != 'hello');
document.write('hello' == 'Hello');
</script>
```

Az eredmény: truefalsefalse .

Két boolean között végezhetünk **ÉS** és **VAGY** műveletet. Az ÉS művelet két és jel jelöli: **&&**. A VAGY műveletet pedig két függőleges vonal: **||**.

```
<script>
document.write(true && false);
document.write(true || false);
document.write(true || 5 < 4);
</script>
```

Az eredmény: falsetruetrue.

false && false == false
true && false == false
false && true == false
true && true == true
false || false == false
true || false == true
false || true == true
true || true == true

Egy boolean-en **NEM** műveletet végezhetünk a felkiáltójellel: **!**.

```
<script>
document.write(!true);
document.write(!false);
</script>
```

!true == false		!false == true

Zárójeleket boolean műveleteknél is használhatunk.

Találkozhatunk olyannal is, hogy három egyenlőségjel **===** van egymás után. A három egyenlőségjel szigorúbb a két egyenlőségjelnél, mert nem csak a két értéknek, hanem a két érték típusának is meg kell egyeznie, például mindkettőnek szövegnek kell lennie.

```
<script>
    document.write(1 == "1");
    document.write("1" === 1);
</script>
```

Az eredmény: truefalse.

A **!==** pedig a **===** ellentéte. Akkor lesz false, ha a **===** true és fordítva.

A **window.confirm** feldob egy ki ablakot, amin OK vagy Cancel gombot nyomhatunk, és boolean-t ad vissza. Az **OK** a **true**, a **Cancel** a **false**.

A window.prompt, window.confirm, window.alert helyett írhatjuk röviden azt is, hogy **promt**, **confirm**, **alert**.

```
<meta charset="utf-8"/>
<script>
    var pontok = 0;
    var nev = prompt("Mi a neved?", "Gizi");
    if ("Domi" == nev) {
```

```
        pontok = pontok + 1;
    }
    var biztos = confirm("Biztos " + nev + " vagy?");
    if (biztos) {
        pontok = pontok + 1;
    }
    alert("Hello " + nev + "!" + "\nPontjaid: " + pontok);
</script>
```

A **\n** egy új sort kezd a szövegben.

UNDEFINED, NULL

Ha egy változónak nem adunk értéket, akkor **undefined** lesz. Ha később ki akarjuk deríteni róla, hogy undefined-e vagy sem, akkor a dupla egyenlőségjellel **==** vagy a felkiáltójel egyenlőséggel **!=** tehetjük meg.

```
<script>
var a;
document.write(a);
document.write(a == undefined);
</script>
```

Az eredmény: undefinedtrue.

A **null** hasonlóan működik, a segítségével kihangsúlyozhatjuk, hogy nem akarunk értéket adni a változónak.

```
<script>
var a = null;
document.write(a);
document.write(a == null);
</script>
```

Az eredmény: nulltrue.

Az **==** vagy **!=** használatakor mindegy, hogy undefined-del vagy null-lal hasonlítjuk össze a változót, ugyanazt az eredmény fogjuk kapni.

```
<script>
var a ;
document.write(a == undefined);
document.write(a == null);
</script>
```

Erdmény: truetrue.

```
<script>
var a = null;
document.write(a == undefined);
document.write(a == null);
</script>
```

Erdmény: truetrue.

A három egyenlőségjel **===** viszont szigorúan megkülönbözteti az **undefined** és a **null** értéket.

```
<script>
var x;
var y = null;
document.write(x == y);
document.write(x === y);
</script>
```

Az erermény: truefalse.

TYPEOF

Ha valaminek meg akarjuk tudni a típusát, arra a **typeof** jó.

```
<script>
var x;
var y = 'valami';
document.write(typeof x);
document.write(typeof y);
document.write(typeof 10);
</script>
```

Eredmény: undefinedstringnumber.

IF

Ha azt akarjuk, hogy a program egy része csak akkor fusson le, ha egy feltétel igaz akkor az **if** parancsot kell használnunk.

```
<script>
if (10 > 5) {
    document.write('nagyobb');
}
if (10 < 5) {
    document.write('kisebb');
}
</script>
```

Az eredmény: nagyobb.

Az **if** után a zárójelbe **( )** olyan kifejezést kell írni, aminek az eredménye **boolean**.

Ha a sima zárójelben **( )** lévő kifejezés igaz vagyis **true**, akkor lefut a kapcsos zárójelben **{ }** lévő programrészlet. A kapcsos zárójelben lévő részt **blokk**nak is szokták hívni. És a blokk belsejét 2 vagy 4 üres hellyel beljebb szokták kezdeni, hogy áttekinthetőbb legyen a program. A blokk több sornyi programból is állhat.

Ha a ( )-be olyan kifejezést írunk, ami **nem boolean**, akkor a blokk akkor fog lefutni, ha a kifejezés nem null és nem undefined.

```
<script>
var x;
var y = 'valami';
if (x) {
    document.write('x');
}
```

```
if (y) {
    document.write('y');
}
</script>
```

Az eredmény: y.

Az **else** azt jelenti, hogy egyébként. Az **else** mögött lévő blokk akkor fut le, ha az **if** utáni blokk nem fut.

```
<meta charset="utf-8"/>
<script>
var kor = window.prompt('Hány éves vagy?');
if (kor < 10) {
    document.write('kicsi vagy');
}
else {
    document.write('nagy vagy');
}
</script>
```

Ha több mint két esetet akarunk vizsgálni, akkor **else if** -et használhatunk:

```
<meta charset="utf-8"/>
<script>
var kor = window.prompt('Hány éves vagy?');
if (kor < 10) {
    document.write('kicsi vagy');
}
else if (kor < 20) {
    document.write('közepes vagy');
}
else {
    document.write('nagy vagy');
}
</script>
```

Az **isNaN** segítségével vizsgálhatjuk meg, hogy valami szám-e. Az isNaN az **is not a number** rövidítése.

```
<meta charset="utf-8"/>
<script>
var szam = window.prompt('Írj be egy számot!');
if (isNaN(szam)) {
    document.write('ez nem is szám');
}
else {
    document.write('ez szám');
}
</script>
```

Az **if else** szerkezethez hasonló a **kérdőjel kettőspont**. Vele viszont csak változóknak adhatunk értéket és csak egy kifejezés lehet a **?** és a **:** után. Az előző példa **? :** segítségével átírva:

```
<meta charset="utf-8"/>
<script>
var szam = window.prompt('Írj be egy számot!');
document.write(isNaN(szam) ? 'ez nem is szám' : 'ez szám');
</script>
```

While, For

A **while** és a **for** segítségével egy blokkot többször is lefuttathatunk. A lenti programban a ++ azt csinálja, hogy eggyel megnöveli a szam változó értékét.

```
<script>
var szam = 0;
while (szam < 3) {
    document.write(szam + ' ');
    szam++;
}
</script>
```

Az eredmény: 0 1 2 .

A **szam++** helyett azt is írhatnánk, hogy **szam = szam + 1**, vagy azt is hogy **szam += 1**. Mind a három ugyanazt csinálja. Ha a szam változó értékét nem növelnénk, akkor a szam örökre kisebb maradna 3-nál, és a program sose futna le.

A **for** a **while**-hoz hasonlóan működik, csak vele egy sorban elintézhetjük a szam kezdeti értékének a megadását, a vizsgálatát, hogy kisebb-e 3-nál és a növelését:

```
<script>
for (var szam = 0; szam < 3; szam++) {
    document.write(szam + ' ');
}
</script>
```

A **while**-t és a **for**-t összefoglaló néven magyarul **ciklus**nak, angolul **loop**-nak hívják.

Array

Az **Array**, magyarul **tömb** arra való, hogy több azonos vagy különböző típusú változót tároljunk benne például számokat vagy szövegeket. Szögletes zárójellel adhatunk neki értéket és kérdezhetjük le egy elmét. Az elemek számozása nullával kezdődik.

```
<script>
var ar = ['blue', 'green', 'yellow'];
document.write(ar[0]);
document.write(ar[2]);
</script>
```

Az eredmény: blueyellow.

Az egész array-t is kiírathatjuk.

```
<script>
var ar = ['blue', 'green', 'yellow'];
document.write(ar);
</script>
```

Az eredmény Firefox böngészővel: blue,green,yellow.

Ha az array egy elemét meg akarjuk változtatni, arra is a szögletes zárójelet használhatjuk.

```
<script>
var ar = ['blue', 'green', 'yellow'];
ar[2] = 'black';
document.write(ar);
</script>
```

Az eredmény: blue,green,black.

Az array-ban található elemek számát a **length** segítségével kérdezhetjük le:

```
<script>
var ar = ['blue', 'green', 'yellow'];
document.write(ar.length);
</script>
```

Az eredmény: 3.

Egy **for ciklus** segítségével az összes elemet és a pozícióját kiírathatjuk.

```
<script>
var ar = ['blue', 'green', 'yellow'];
for (var i = 0; i < ar.length; i++) {
    document.write(i + ':' + ar[i] + ' ');
}
</script>
```

Az eredmény: 0:blue 1:green 2:yellow

A **join** segítségével összefűzhetjük az elemeket egy szöveggé. Megadhatunk egy elválasztó szöveget is. A join magát az array-t nem fogja megváltoztatni, a join művelet eredménye lesz az összefűzött szöveg.

```
<script>
var ar = ['blue', 'green', 'yellow'];
document.write(ar.join(' * '));
</script>
```

Az eredmény: blue * green * yellow .

Új elemet hozzáadni a **push** segítségével tudunk.

```
<script>
var ar = ['blue', 'green', 'yellow'];
ar.push('pink');
document.write(ar[3]);
</script>
```

Az eredmény: pink.

A **splice** segítségével törölhetünk elemeket egy array-ból. Meg kell adni, hogy melyik pozíciótól kezdve hány elemet akarunk törölni.

```
<script>
var ar = ['blue', 'green', 'yellow'];
```

```
ar.splice(0, 1);
document.write(ar);
</script>
```

Az eredmény: green,yellow.

FUNCTION

A **function** (magyarul: függvény) segítségével olyan programrészt írhatunk, amit többször is meghívhatunk.

A function-nak szoktunk nevet adni, ami a lenti példában hello. Utána két sima kerek zárójel **()**, utána pedig két kapcsos zárójel **{}** közé írjuk a blokkot amit futtatni akarunk.

A function futtatásához a nevét és a zárójeleket kell leírni. A lenti példában kétszer is futtatjuk, más néven meghívjuk a function-t.

```
<script>
function hello() {
    document.write('hello');
}
hello();
hello();
</script>
```

Az eredmény: hellohello.

A function a **return** segítségével vissza is adhat egy értéket a meghívójának.

```
<script>
function szaz() {
    return 100;
}
document.write(szaz());
</script>
```

Az eredmény: 100.

A function fogadhat is értékeket a meghívójától. Ezeket **paraméter**eknek hívják.

A lenti példában a=1 és b=2 lesz.

```
<script>
function osszead(a, b) {
    return a + b;
}
document.write(osszead(1, 2));
</script>
```

Az eredmény: 3.

A profi JavaScript programozók szinte mindent function-ökbe tesznek, és sok rövid function-ből építik fel a programjukat.

OBJECT

Az **object** (magyarul: objektum) egy összetett változó, melybe más változókat, akár **array**, **function** vagy **object** típusú változókat is tehetünk. A program blokkokhoz hasonlóan kapcsos zárójelet {} kell használni egy új **object** megadásához.

Az objektum elemeinek nevet adunk, a név és az érték közé kettőspontot **:** teszünk, a név-érték párokat vesszővel **,** választjuk el egymástól.

És pont **.** vagy szögletes zárójel **[]** segítségével kérdezhetjük le az értéküket. A pontot akkor szoktuk használni, ha már a program írásakor pontosan tudjuk, melyik mezőre vagyunk kíváncsiak, szögletes zárójelet pedig akkor, ha a mező neve csak a program futásakor fog kiderülni, mert például a mező neve is egy változó.

```
<script>
var obj = {
   szam: 13,
   szoveg: 'hello'
}
document.write(obj.szoveg + obj.szam);
document.write(obj['szoveg'] + obj['szam']);
</script>
```

Az eredmény: hello13hello13.

A pont vagy a szögletes zárójel segítségével meglévő elemet is módosíthatunk és új elemet is vehetünk fel.

```
<script>
var obj = {
   szam: 13,
   szoveg: 'hello'
}
obj.most = new Date();
document.write(obj.most);
</script>
```

A **new Date()** a mai dátumot adja vissza.

Az **Object.keys** segítségével megnézhetjük, milyen mezői vannak egy **object**-nek.

```
<script>
var obj = {
   szam: 13,
   szoveg: 'hello'
}
document.write(Object.keys(obj));
</script>
```

Az eredmény: szam,szoveg.

A **for** ciklusnak van egy speciális formája, a **for in**, ami segítségével végigmehetünk az **object** mezőin.

A lenti példában a prop változóba fog kerülni a mező neve. A mező értékét most nem ponttal, hanem szögletes zárójellel érjük el: **obj[prop]**, mert az csak a program futásakor derül ki, hogy az objektum melyik mezőjét olvassuk be.

```
<script>
var obj = {
   szam: 13,
   szoveg: 'hello'
}
for (var prop in obj) {
   document.write(prop);
   document.write(obj[prop]);
}
</script>
```

Az eredmény: szam13szoveghello.

DATE

Már láttuk, hogy a **new Date()** a mai dátumot és pontos időt adja vissza. Ennek a részeit (év, hónap, nap, óra, perc, másodperc, ezredmásodperc) **get** function-ökkel kérdezhetjük le (a functiont method-nak vagy magyarul metódusnak is szokták nevezni).

```
<script>
var most = new Date();
document.write(most + '<br>');
document.write(most.getFullYear() + '<br>');
document.write(most.getMonth() + '<br>');
document.write(most.getDate() + '<br>');
document.write(most.getHours() + '<br>');
document.write(most.getMinutes() + '<br>');
document.write(most.getSeconds() + '<br>');
document.write(most.getMilliseconds() + '<br>');
document.write(most.getTime() + '<br>');
document.write(most.getDay() + '<br>');
</script>
```

Az eredmény:

Tue Aug 21 2018 20:18:31 GMT+0200 (Central Europe Daylight Time)
2018
7
21
20
18
31
684
1534875511684
2

A **getMonth()** a hónapot nullával kezdődően számozza, például 0 = január.

A **getTime()** az 1970. január 1. óta eltelt ezredmásodperceket adja vissza. Akkor jó, ha két egymáshoz közeli időpontot akarunk egymással összehasonlítani.

A **getDay()** a hét aktuális napját adja vissza a számozás 0-val kezdődik és a 0 = vasárnap.

A dátumot beállítani set metódusokkal tudjuk. Ezek ugyanazok, mint a get metódusok, csak nincs setDax().

A következő példa 2000. január 1-re állítja a dátumot, de az óra-percet nem bántja.

```
<script>
var most = new Date();
most.setFullYear(2000);
most.setMonth(0);
most.setDate(1);
document.write(most);
</script>
```

Az eredmény: Sat Jan 01 2000 20:34:06 GMT+0100 (Central Europe Standard Time).

Ha az óra-percet is nullázni akarjuk, egyszerűbb megoldás is van:

```
<script>
var most = new Date(2000, 0, 1);
document.write(most);
</script>
```

Az eredmény: Sat Jan 01 2000 00:00:00 GMT+0100 (Central Europe Standard Time).

A new Date() -nek vesszővel elválasztva megadhatunk óra-perc-másodperc-ezredmásodpercet is:

new Date(év, hónap, nap, óra, perc, másodperc, ezredmásodperc).

A paraméterek listájából a jobboldalról akárhányat elhagyhatunk, akkor az nullázódik.

```
<script>
var most = new Date(1999, 11);
document.write(most);
</script>
```

Az eredmény: Wed Dec 01 1999 00:00:00 GMT+0100 (Central Europe Standard Time).

MATH

A **Math.random()** egy 0 és 0.9999999999999999 közötti véletlen számot ad vissza.

```
<script>
document.write(Math.random());
</script>
```

Az eredmény valami ilyesmi lesz: 0.4757217074027479.

A **Math.floor()** egy számról levágja a tizedesvessző (a programozásban és az angolban tizedespont) utáni számjegyeket.

```
<script>
document.write(Math.floor(12.66));
</script>
```

Az eredmény: 12.

Az előző kettő segítségével csinálhatunk tetszőleges véletlen számokat. Az alábbi program például 1 és 10 közötti véletlen számokat gyárt.

```
<script>
document.write(Math.floor(Math.random() * 10) + 1);
</script>
```

Először a Math.random() * 10 csinál egy 0 és 9.9999999999999999 közötti számot, a Math.floor() egy 0 és 9 közötti egész számot, amihez ha 1-et hozzáadunk, akkor egy 1 és 10 közötti egész számot kapunk.

MŰVELETEK VÉGREHAJTÁSI SORRENDJE

A lenti táblázat a különböző műveletek végrehajtási sorrendjét (angolul: operator percedence) tartalmazza.

Felül a magasabb szinten vannak azok a műveletek, amik először hajtódnak végre, és lefelé haladva a műveletek egyre később hajtódnak végre.

A táblázatban az is látszik, hogy ha két műveletnek azonos a szintje, akkor balról jobbra vagy jobbról balra hajtódnak végre.

A táblázat elsősorban azokat a műveleteket tartalmazza, melyek szerepelnek ebben a tananyagban.

Szint	Művelet	Leírás	Irány
12	( )	zárójel	-
11	obj. arr[ ] func()	object elemének elérése array elemének elérése function meghívása	balról jobbra
10	-szam ++szam --szam typeof	negatív szám eggyel növelés eggyel csökkentés változó vagy érték típusa	jobbról balra
9	* / %	szorzás osztás egész számok osztásakor a maradék	balról jobbra
8	+ -	összeadás kivonás	balról jobbra
7	< <= > >=	kisebb kisebb egyenlő nagyobb nagyobb egyenlő	balról jobbra
6	== != === !==	egyenlőség nem egyenlőség szigorú egyenlőség szigorú nem egyenlőség	balról jobbra
5	&&	logikai ÉS (ha a bal oldal false, akkor a jobb ki sem értékelődik)	balról jobbra
4	||	logikai VAGY (ha a bal oldal true, akkor a jobb ki sem értékelődik)	balról jobbra

3	? :	ha akkor egyébként	jobbról balra
2	= += -= *= /=	értékadás értékhez hozzáadás értékből kivonás érték szorzása érték osztása	jobbról balra
1	,	vessző	balról jobbra

A teljes táblázat angol nyelven ezen a két weboldalon található meg:

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

https://www.w3schools.com/js/js_arithmetic.asp

Most pedig gyakorlásképpen nézzünk néhány példát. Az eredmények a programba vannak kommentelve.

```
<script>
document.write(10 + 1 - 2 + 3);                // 12
document.write(10 + 1 - (2 + 3));              // 6
document.write(10 + 2 * 2);                    // 14
document.write((10 + 2) * 2);                  // 24
document.write(10 > 5 && 2 > 1);               // true
document.write(10 > 5 == 2 > 1);               // true
document.write(true || false && false);        // true
document.write((true || false) && false);      // false
document.write(typeof 10 + 5);                 // number5
document.write(!true == false);                // true
document.write(10 > 5 ? 1 + 2 : 1);            // 3
document.write(2 * (10 + 10));                 // 40
document.write('hello' != 'hello');            // false
document.write('hello' != 'hello' == false);   // true
</script>
```

IDŐZÍTÉS

A **window.setTimeout()** segítségével meghívhatunk egy **function**-t nem azonnal, hanem adott idő eltelte után. Az function nevét és ezredmásodpercben a késleltetési időt kell megadni. A lenti program 3 másodperc múlva hívja meg a hello nevű function-t.

```
<script>
function hello() {
    window.alert('hello')
}
window.setTimeout(hello, 3000);
</script>
```

A **window.setInterval()** folyamatosan hívogatja a megadott nevű function-t ezredmásodpercben megadott időnként. A **window.clearInterval()** segítségével tudjuk leállítani. Mivel egyszerre több interval is futhat, a clearInterval()-nak át kell adni azt a változót, amelyikben az interval-t csináltuk, hogy el tudja dönteni, melyiket állítsa le.

A lenti példa 1 másodperces késleltetéssel elszámol 10-ig. A document.write() sajnos nem működik jól együtt a window.setInterval() -lal, ezért most a konzolra írjuk az eredményt, szóval a tananyag elején leírtak szerint nyissunk egy konzolt.

```
<script>
var x = 1;
var inter = window.setInterval(kiir, 1000);
function kiir() {
    console.log(x++);
    if (x > 10) {
        window.clearInterval(inter);
    }
}
</script>
```

Az **x++** eggyel növeli x értékét, de csak miután kiírtuk a konzolra. Ha előtte akarnánk növelni, akkor használhatjuk az **++x** kifejezést. Próbáljuk ki!

3 DOM ALAPOK

A DOM a Document Object Model rövidítése. A DOM a kapcsolat a HTML oldal és a rajta található JavaScript program között, a segítségével tudjuk JavaScript-ből módosítani a HTML oldalt. Magának a HTML oldalnak és a benne található összes elemnek van egy megfelelője a DOM-ban, amivel piszkálhatjuk.

Ebben a részben a konzolra fogjuk írni az eredményeket **console.log**-gal, mert a **document.write** azzal indul, hogy törli a DOM-ot. Szóval kapcsoljuk be a böngésző konzolját!

LOCATION

A **window.location** visszaadja a weboldal **URL**-jét. Az URL az a cím ami a böngésző fenti széles mezőjében látszik.

```
<script>
document.write(window.location);
</script>
```

Az eredmény valami ilyesmi lesz: file:///C:/Bubuka/BubukaProgramjai/1.html

A BÖNGÉSZŐABLAK SZÉLESSÉGE, MAGASSÁGA

Ha egy HTML oldalt akarunk JavaScript-ből piszkálni, nem árt tudni, mennyi helyünk van a böngészőablakban. A **window.innerWidth** adja meg a szélességét, a **window.innerHeight** pedig a magasságát.

```
<script>
console.log('szelesseg', window.innerWidth);
console.log('magassag', window.innerHeight);
</script>
```

Az eredmény a konzolon valami ilyesmi lesz: szelesseg 1314 magassag 399

Egy HTML elem megkeresése a HTML oldalon id alapján

Ha a HTML elemnek adunk **id**-t, akkor a **document.getElementById()** segítségével könnyen rátalálhatunk az elemre.

```
<meta charset="utf-8"/>
<div id="elso">1.</div>
<div id="masodik">2.</div>
<script>
var elem = document.getElementById("masodik");
console.log(elem);
</script>
```

Valami ilyesmi jelenik meg a konzolon: .

Ha a konzolban egérrel ráklikkelünk -re, akkor megjelennek a tulajdonságai (angolul ez most: attribute). Keressük meg ezt a attribute-ot: **innerHTML**.

Egy HTML elem módosítása: innerHTML

Most pedig kérdezzük le és írjuk át az **innerHTML**-t.

```
<meta charset="utf-8"/>
<div id="elso">1.</div>
<div id="masodik">2.</div>
<script>
var elem = document.getElementById("masodik");
console.log(elem.innerHTML);
elem.innerHTML = '3.';
</script>
```

A konzolon megjelenik a 2., mert azt még a DOM elem átírása előtt írattuk ki. A HTML oldalon viszont már a 2. helyett 3. látszódik. Az innerHTML-ben nem csak szöveget, hanem más HTML elemeket is megadhatunk, pl. **elem.innerHTML = '3.';**

Egy HTML elem tulajdonságai

Egy HTML elemnek az **.id** tulajdonsága visszaadja az elem **id**-jét (micsoda meglepetés ☺).

A **.tagName** az elem típusát, pl. DIV.

Az **.innerHTML**-lel már találkoztunk, az elemben lévő HTML tartalmat adja vissza.

A **.textContent** pedig a benne levő szöveget adja vissza HTML elemek nélkül.

```
<div id="elso">Hello <b>Cinc</b>!</div>
<script>
   var elem = document.getElementById("elso");
   console.log(elem.id);
   console.log(elem.tagName);
   console.log(elem.innerHTML);
   console.log(elem.textContent);
</script>
```

Az eredmény:

```
elso
DIV
Hello <b>Cinc</b>!
Hello Cinc!
```

Egy HTML elem megkeresése a HTML oldalon: querySelector

A **document.querySelector()** segítségével a CSS-hez hasonló módon kereshetünk meg egy HTML elemet **típus**, **id** vagy CSS **class** alapján.

A lenti példában először a HTML elem típusára (div), aztán az id-jére (masodik), majd a CSS class-szára (class2) keresünk.

```
<meta charset="utf-8"/>
<div id="elso" class="class1">1.</div>
<div id="masodik" class="class2">2.</div>
<script>
var elem = document.querySelector('div');
console.log(elem);
elem = document.querySelector('#masodik');
console.log(elem);
elem = document.querySelector('.class2');
console.log(elem);
</script>
```

Az eredmény:

```
<div id="elso" class="class1">
<div id="masodik" class="class2">
<div id="masodik" class="class2">
```

CSS style

Egy HTML elem CSS stílusát az elem **style.cssText** segítségével írhatjuk át vagy kérdezhetjük le. Ez olyan, mintha HTML-ben egy **style** tulajdonságot adnánk egy HTML elemhez.

A lenti példában egy 50-szer 50-es piros négyzetként jelenítjük meg a kiválasztott **div** elemet, majd a frissen beállított stílust kiírjuk a konzolra.

```
<meta charset="utf-8"/>
<div>1.</div>
<script>
var elem = document.querySelector('div');
elem.style.cssText = 'background-color:red; height:50px; width:50px;';
console.log(elem.style.cssText);
</script>
```

Ha csak egy CSS tulajdonságot (angolul ez most: property) akarunk átírni, használhatjuk **style.setProperty()**-t is. Ha egy CSS tulajdonságot akarunk törölni, akkor **style.removeProperty()**.

A lenti példában egy pontozott (angolul: **dotted**) keretet (angolul: **border**) rakunk a **div** köré. Ezután töröljük a HTML-ben korábban megadott szélességét (angolul: **width**), amitől a szélessége ki fogja tölteni a teljes böngészőablakot.

```
<meta charset="utf-8"/>
<div style="height:50px; width:50px;">1.</div>
<script>
var elem = document.querySelector('div');
elem.style.setProperty('border', 'dotted');
elem.style.removeProperty('width');
</script>
```

4 ESEMÉNYKEZELÉS

ONMOUSEENTER, ONMOUSELEAVE

A tananyag elején már megismerkedhettünk az onclick eseménnyel, most először az **onmouseenter** (egér belép) és az **onmouseleave** (egér távozik), majd később az onkeypress (billentyű megnyom) eseménnyel fogunk megismerkedni.

A következő példában egy div elemre rakunk két eseménykezelőt, amik akkor hívódnak meg, ha az egeret a div fölé húzzuk vagy elhúzzuk onnan.

```
<meta charset="utf-8"/>
Húzd az egeret a négyzetbe, majd húzd ki belőle!
<div style="border: solid blue; width: 100px; height: 100px;"
    onmouseenter="mouseenter()"
    onmouseleave="mouseleave()">
</div>
<script>
   var elem = document.querySelector('div');
   function mouseenter() {
       elem.style.setProperty('background', 'yellow');
   }
   function mouseleave() {
       elem.style.removeProperty('background');
   }
</script>
```

ONKEYPRESS, EVENT.WHICH

Az **onkeypress** esemény akkor hívódik meg, ha egy kiválasztunk egy HTML elemet és megnyomunk egy gombot a billentyűzeten.

A lenti példában az eseménykezelőt a **body** elemre tesszük, így elég bárhova klikkelni a HTML oldalon, és már ki is választottuk a body elemet.

Ha a meghívott **function**-nak átadunk egy tetszőleges nevű paramétert, abba a böngésző bele fogja tenni az esemény (angolul: **event**) tulajdonságait. Ezek közül a tulajdonságok közül a **which** tulajdonságot fogjuk használni, ami a megnyomott billentyű kódját fogja visszaadni.

```
<meta charset="utf-8"/>
<html>
   <body onkeypress="gomb(event)">
       Klikkelj valahova a HTML oldalon, utána nyomj meg gombokat a billentyűzeten!
       <div></div>
       <script>
```

```
            var elem = document.querySelector('div');
            function gomb(ev) {
                elem.innerHTML = ev.which;
            }
        </script>
    </body>
</html>
```

ONLOAD

Az **onload** eseménykezelőt a body elemre szokták tenni, akkor hívódik meg, ha a HTML oldal már az összes képpel együtt teljesen betöltődött.

A lenti példa bekéri a felhasználó (angolul: user) nevét, és attól függően köszönti, hogy hány óra van.

```
<meta charset="utf-8"/>
<script>
function ido() {
    nev = window.prompt("Neved?","vendég");
    datum = new Date();
    ido = datum.getHours();
    if (ido>23 || ido<5) {
        document.write("Jó éjt "+nev+"!");
    } else {
        document.write("Jó napot "+nev+"!");
    }
}
</script>
<body onload="ido();">
</body>
```

EVENT.TARGET

A következő példában egy div-re teszünk egy **onclick** eseménykezelőt. A **click** esemény egyébként olyan, mintha egy **mousedown** és egy **mouseup** esemény történne egymásután.

Az esemény (event) **.target** tulajdonsága visszaadja azt a HTML elemet, amelyiken az esemény történt. A visszakapott HTML elemnek lekérdezhetjük a tulajdonságait.

Most is a konzolra írjuk majd az eredményt, szóval a tananyag elején leírtak szerint nyissunk egy konzolt.

```
<div id="d" style="background: pink;" onclick = "klikk(event);">
Klikkelj <b id="b">ide</b>!
</div>
<script>
    function klikk(event) {
        console.log(event.target.id);
        console.log(event.target.innerHTML);
    }
</script>
```

Ha az **ide** szövegre klikkelünk, az eredmény:

b
ide

Ha a **Klikkelj** szövegre klikkelünk, az eredmény:

d
Klikkelj ide!

Az esemény (event) **.currentTarget** tulajdonsága pedig azt a HTML elemet adja vissza, amire az eseménykezelőt tettük. Tehát az akkor is a **div**-et adja vissza, ha a **b**-n belülre klikkelünk.

5 KOCKA

Ebben a részben nem nagyon fogunk új dolgokat tanulni, inkább az eddig tanultakból fogunk építkezni. Egy HTML **div** elemet fogunk CSS segítségével kicsit feldíszíteni, és azzal fogunk játszani.

Ha nagyon precízek akarunk lenni, itt igazából négyzeteket fogunk rajzolni, de beláthatjuk, hogy egy kockát is láthatunk olyan szögből, hogy négyzetnek tűnjön.

BUTTON

A lenti példában egy **div** elemet formázunk meg **style** segítségével. Továbbá kiteszünk a HTML oldalra két nyomódombot (button). Mindkettő az **onclick** eseménykezelőt használja. Az egyik gomb 50-nel csökkenti, a másik 50-nel növeli x értéket, majd mindkét nyomógomb beállítja a div **style.left** (a div baloldala hány pontra legyen a böngészőablak bal szélétől) tulajdonságát x-re.

```
<div id="div1"></div>
<style>
   div {
      position:absolute;
      top:50; left:50;
      height:50;width:50;
      background:yellow;
      border:solid 8px red
   }
</style>
<script>var x=50;</script>
<button onclick="x=x-50;document.getElementById('div1').style.left=x">&lt;&lt;</button>
<button
onclick="x=x+50;document.getElementById('div1').style.left=x">&gt;&gt;</button>
```

Ötlet: a **background-image** CSS tulajdonsággal bármilyen képet be lehet tenni a **div** mögé háttérnek. Például ha van a .html fájllal azonos könyvtárban egy kep.jpg fájlunk, azt így tehetjük be a div mögé:

```
div {
   background-image: url("kep.jpg");
}
```

KEY

A következő példa az **onkeypress** eseménykezelőt használja. A 4-es gomb megnyomására balra, a 6-os gomb megnyomására jobbra mozdítja a kockát. Azért 4 és 6, mert a billentyűzet

jobb oldalán a numerikus billentyűzeten, ha van olyan, a 4 a balra nyíl és a 6 a jobbra nyíl. A 4-es gomb kódja 52, a 6-osé 54.

```
<html>
<meta charset="utf-8"/>
Irányítás ezekkel a gombokkal: 4, 6.
<body onkeypress="gombnyom(event);">
   <style>
      #div1 {
         position:absolute;
         top:50; left:200;
         height:80; width:80;
         background-color:yellow;
         border-style:solid; border-width:10; border-color:red;
         display:block
      }
   </style>
   <div id="div1"></div>
   <script>
   var x = 200;
   function gombnyom(event) {
      if (event.which == 52) {
         x = x – 10;
      }
      if (event.which == 54) {
         x = x + 10;
      }
      document.getElementById('div1').style.left = x;
   }
   </script>
</body>
</html>
```

Ha **console.log** segítségével kiírjuk a konzolra az **event.which** értékét, akkor bármelyik gombnak megtudhatjuk a kódját.

Feladat: Csináljuk meg, hogy a 2-es gombbal lefelé, a 8-assal felfelé is mozogjon a kocka. A 2-es kódja 50, a 8-asé 56. Szükségünk lesz egy y változóra és a **style.top** CSS tulajdonságot kell beállítani.

MOUSE

A következő példában az **onmousemove** és az **onclick** eseménykezelőket fogjuk használni. Ha a div elem fölé helyezzük az egeret és jobbra-balra mozgatjuk vagy klikkelünk, akkor valami történni fog.

Jobbra-balra mozgatáskor az **event.clientX** segítségével kitaláljuk, hogy hol történt az esemény, vagyis hova mozdult el az egér. Azt szeretnénk, hogy ott legyen a kocka közepe, ahova az egeret mozgatjuk. Viszont a kockának csak azt tudjuk megmondani a **left** segítségével, hogy hol legyen a baloldala. A kocka baloldala fél kockányival balra van a kocka közepétől. A kocka szélessége 70 pixel (border-width + width + border-width). Ezért állítjuk a kocka bal szélét **event.clientX – 70/2** értékre. Ezt az értéket először beállítjuk a szokásos módon: **style.left**, majd az **innerHTML** segítségével bele is írjuk a kockába.

Klikkeléskor a **style.background** segítségével átállítjuk a kocka színét. Ha sárga volt, akkor zöldre. Ha zöld volt, akkor sárgára.

```
<html>
   <meta charset="utf-8"/>
   Menj az egérrel a kocka fölé, mozgasd az egeret jobbra-balra és klikkelj!
   <style>
     #div1 {
          position:absolute;
          top:50;
          left:200;
          height:50;
          width:50;
          background-color:yellow;
          border-style:solid;
          border-width:10;
          border-color:red;
          display:block
     }
   </style>
   <div id="div1" onmousemove="mozgat(event);" onclick="szinez()"></div>
   <script>
       var x=200;
       function mozgat(event) {
          x = event.clientX – 70/2;
          document.getElementById('div1').style.left = x;
          document.getElementById('div1').innerHTML = "x=<b>" + x + "</b>";
       }
       var szin = "yellow";
       function szinez() {
          if (szin == "yellow") {
             szin = "#88FF88";
          } else {
             szin = "yellow";
          }
          document.getElementById('div1').style.background = szin;
       }
   </script>
</html>
```

TIMER

A lenti programban az **onload** eseménykezelő a HTML oldal betöltődésekor meghívja a mozgat() function-t. A mozgat() function hússzal növeli a **style.left**-et, amitől a kocka jobbra indul el. A function végén van egy **window.setTimeout()** hívás, ami 1 tizedmásodperc késleltetéssel újra meghívja a mozgat() function-t, vagyis saját magát. A saját magukat meghívó programokat **rekurzív program**oknak hívják.

A mozgat() function futás közben azt is vizsgálja, hogy nem szalad-e ki a böngészőablakból a kocka. Ha kiszaladna, akkor visszafordítja és onnantól kezdve nem növeli, hanem csökkenti a **style.left**-et, amitől a kocka balra indul el. A képernyő bal szélén pedig ismét megfordítja. A 70 a kocka teljes szélessége. A 20 pedig egy biztonsági sáv az ablak széle körül, amibe beleérve

visszafordítjuk a kockát. Mivel a kocka minden lépésben 20-at halad, így kicsit se tud kimenni a képernyőről.

```
<html>
<body onload="mozgat();">
   <style>
      #div1 {
         position:absolute;
         top:50;
         left:200;
         height:50;
         width:50;
         background-color:yellow;
         border-style:solid;
         border-width:10;
         border-color:red;
        display:block
      }
   </style>
   <div id="div1"></div>
   <script>
      var x = 200;
      var elore = true;
      function mozgat() {
         if (x < 20) {
            elore = true;
         }
         if (x > window.innerWidth - 70 - 20) {
            elore = false;
         }
         if (elore) {
            x += 20;
         } else {
            x -= 20;
         }
         document.getElementById('div1').style.left = x;
         document.getElementById('div1').innerHTML = "x=<b>" + x + "</b>";
         window.setTimeout("mozgat();", 100);
      }
   </script>
</body>
</html>
```

RANDOM

A lenti program 50-esével végigmegy balról jobbra a böngészőablakon és mindenhova kitesz véletlen magasságba: **top** egy véletlen színű: **background-color** kockát.

A **top**-ba az y változó kerül, ami egy véletlen szám 0-tól az ablak magassága – 50 ig. Az ablak magassága: **window.innerHeight**. A – 50 azért kell, mert a kocka 50 pixel magas.

A kocka színéhez 3 véletlen számot generálunk, mindhárom 0 és 255 közötti, és a szín vörös, zöld és kék tartalmát adják meg.

Az x, y, color változók felhasználásával összerakunk egy olyan szöveget, ami egy **div** elemet tartalmaz, majd ezt a szöveget a **document.write** segítségével írjuk bele a HTML oldalba.

Annyi érdekessége van a dolognak, hogy a böngésző igazából nem azonnal hajtja végre a **document.write**-okat, hanem összevárja és egyszerre jeleníti meg őket, miután a programunk már lefutott.

A document.write-tal kezdődő sor itt csak két sorban fért ki, de a programban egy sorba kellene írni őket egészen a második sor pontosvesszőjéig.

```
<script>
    for (var x = 0; x <= window.innerWidth - 50; x = x + 50) {
        var y = Math.floor(Math.random() * (window.innerHeight – 50));
        var red = Math.floor(Math.random() * 256);
        var green = Math.floor(Math.random() * 256);
        var blue = Math.floor(Math.random() * 256);
        var color = "rgb(" + red + "," + green + "," + blue + ")";
        document.write("<div style='position:absolute;top:" + y + ";left:" + x +
";height:50;width:50;background-color:" + color + ";'></div>");
    }
</script>
```

ARRAY

Most egy **array**-ba teszünk öt **object**-et. Minden **object** egy kocka x és y pozícióját és a színét tartalmazza. Ezután egy **for** ciklussal végigmegyünk az array object-jein és kinyerjük az adatokat belőlük az x, y, color változókba. Majd a már ismert módon megjelenítjük a kockákat.

A document.write-tal kezdődő sor itt is csak két sorban fért ki, de a programban egy sorba kellene írni őket egészen a második sor pontosvesszőjéig.

```
<script>
    var negyzetek = [
        {x: 200,  y: 200, szin: '#4637F1'},
        {x: 400,  y: 300, szin: 'red'},
        {x: 600,  y: 50,  szin: 'rgb(15,255,15)'},
        {x: 800,  y: 350, szin: "#006"},
        {x: 1000, y: 100, szin: "yellow"}
    ];
    for (var i = 0; i < negyzetek.length; i++) {
        var x = negyzetek[i].x;
        var y = negyzetek[i].y;
        var color = negyzetek[i].szin;
        document.write("<div style='position:absolute;top:" + y + ";left:" + x +
";height:50;width:50;background-color:" + color + ";'></div>");
    }
</script>
```

6 ÚJ HTML ELEM KÉSZÍTÉSE MÁSKÉPP

MEGLÉVŐ HTML ELEM LÁTHATÓSÁGA

A profik egyáltalán nem szoktak **document.write**-ot használni. Inkább HTML-ben kirakják az összes elemet, amire szükség lehet később, és azt amelyikre éppen nincs szükség, láthatatlanra állítják a **visibility** (magyarul: láthatóság) CSS tulajdonsággal. Ennek két értéke lehet: **visible** (látható), **hidden** (elrejtett).

Ehhez hasonló a **display** CSS tulajdonság is, lehetséges értékei: **block** (látszik), **none** (nem). Erre mindjárt látunk majd példát.

TÁBLÁZAT

A lenti példa egy HTML oldal, ahol a kedvenc kajáimat sorolom fel egy táblázatban. Ha a táblázat alatt a „Többet!” linkre klikkelünk, új sorok jelennek meg a táblázatban és a link átíródik „Kevesebbet!” linkre. A „Kevesebbet!” linkre klikkelve eltűnnek az új sorok.

A program a **style.display** értékeivel játszik. Ahol látszik valami, ott **block** az értéke, ahol nem, ott pedig **none**. De igazából mindkét táblázat és mindkét link folyamatosan kinn van a HTML oldalon, csak néha nem látszódnak.

```
<html>
   <meta charset="utf-8"/>
   <body>
      <b>A kedvenc kajáim:</b><br><br>
      <table class="table1" id="table1" style="display:block;">
         <tr><td>1  </td><td>Kukorica</td></tr>
         <tr><td>2</td><td>Borsó</tr>
         <tr><td>3</td><td>Dinnye</td></tr>
         <tr><td>4</td><td>Húsleves</td></tr>
         <tr><td>5</td><td>Tenger gyümölcsei pizza</td></tr>
      </table>
      <table class="table1" id="table2" style="display:none;">
         <tr><td>1  </td><td>Kukorica</td></tr>
         <tr><td>2</td><td>Borsó</tr>
         <tr><td>3</td><td>Dinnye</td></tr>
         <tr><td>4</td><td>Húsleves</td></tr>
         <tr><td>5</td><td>Tenger gyümölcsei pizza</td></tr>
         <tr><td>6</td><td>Drazsé</tr>
         <tr><td>7</td><td>Lecsó</tr>
         <tr><td>8</td><td>Szilvásgombóc</tr>
         <tr><td>9</td><td>Csokis keksz</td></tr>
         <tr><td>10</td><td>Szőlő</td></tr>
      </table>
      <a id="a1" onClick="a1click();" style="display:none;" href="#">Kevesebbet!</a>
      <a id="a2" onClick="a2click();" style="display:block;" href="#">Többet!</a>
      <script>
         function a1click() {
            document.getElementById('table2').style.display='none';
            document.getElementById('table1').style.display='block';
            document.getElementById('a1').style.display='none';
```

```
            document.getElementById('a2').style.display='block';
        }
        function a2click() {
            document.getElementById('table1').style.display='none';
            document.getElementById('table2').style.display='block';
            document.getElementById('a2').style.display='none';
            document.getElementById('a1').style.display='block';
        }
    </script>
  </body>
</html>
```

ÚJ HTML ELEM HOZZÁADÁSA A DOM-HOZ

Az is egy megoldás, hogy az új elemet hozzáadják a DOM-hoz. A lenti példában egy **hello** elemet készítünk el, és ezt beletesszük a div1 id-jű **div**-be.

```
<div id="div1">
<script>
  var elem = document.createElement('b');
  elem.textContent = 'haha';
  document.getElementById('div1').appendChild(elem);
</script>
```

Ugyanezt az **innerHTML** segítségével is megtehetjük.

```
<div id="div1">
<script>
  document.getElementById('div1').innerHTML = '<b>haha</b>';
</script>
```

7 HANGOK

Töltsünk le egy **.mp3** zene fájlt valahonnan, például innen: http://szilagyidonat.hu/tananyag.html

A letöltött .mp3 fájlt másoljuk a .html fájlunk mellé.

A lenti programban írjuk át az .mp3 fájl nevét, ha más fájlt töltöttünk le.

A DOM-ban egy **audio** típusú elemként jelenik meg a zene fájl, amit a **.play()** segítségével játszhatunk le.

```
<audio id="audio1" preload="auto">
  <source src="lencsilany.mp3" />
</audio>
<script>
  var a1 = document.getElementById('audio1');
  a1.play();
</script>
```

Ha azt akarjuk, hogy a zene lejátszása után mindig újra lejátssza, akkor még a **.play()** előtt ezt kell beállítani:

```
a1.loop = true;
```

A következő példa a Play div-re klikkelve lejátssza az mp3.fájlt, a Pause megállítja a lejátszást, a Rewind pedig visszaáll a zene elejére.

```
<audio id="audio1" preload="auto">
    <source src="lencsilany.mp3" />
</audio>
<div onclick="play()">Play</div>
<div onclick="pause()">Pause</div>
<div onclick="rewind()">Rewind</div>
<style>
    div {
        border: solid;
        width: 70px;
        margin: 10px;
    }
</style>
<script>
    var a1 = document.getElementById('audio1');
    function play() {
        a1.play();
    }
    function pause() {
        a1.pause();
    }
    function rewind() {
        a1.currentTime = 0;
    }
</script>
```

Megjegyzés: Egy weboldalon illik megkérdezni a felhasználót, hogy akar-e zenét, mielőtt bekapcsoljuk.

8 CANVAS

Ebben a részben a példaprogramokba is írtam kommenteket, melyek segíthetik a programok megértését.

A **canvas** egy rajzasztal, amire bármit rajzolhatunk.

A következő példa négyzet, szöveget és egy képet rajzolja ki a **canvas**-ra.

Szükség lesz egy kicsi képre, amit magunk is megrajzolhatunk vagy letölthetünk valahonnan, például innen: http://szilagyidonat.hu/tananyag.html. A példában a potty.png van, amit át kell írni arra a képre, amit használunk.

A lenti példában egy **canvas** elemet készítünk, aminek beállítjuk a stílusát. Ezután az img változóban a **new Image()** segítségével csinálunk egy kép object-et, aminek az **.src** tulajdonságát beállítjuk arra a képre, amit később meg akarunk jeleníteni. A rajzolás a draw() function-ban történik, amit a body **onload** eseménykezelő hív meg. Megkeressük a DOM-ban a canvas elemet és lekérünk tőle egy context-et a **.getContext('2d')** segítségével, aminek azt is

megmondjuk, hogy 2 dimenziós rajzot fogunk készíteni. Ha 3 dimenzióban rajzolnánk akkor **.getContext('webgl')** -t kellene használnunk, de azzal most nem foglalkozunk.

Először egy piros téglalapot rajzolunk, amihez a **.fillStyle** segítségével beállítjuk, hogy milyen színnel legyen kitöltve. A **.fillRect** rajzolja ki a téglalapot a bal felső sarok x és y koordinátájának, majd a jobb alsó sarok x és y koordinátájának a megadásával. Ezután egy félig átlátszó kék téglalap jön. Az átlátszóságot (angolul: transparency) a **.fillStyle** negyedik paramétere adja meg, 0 és 1 közötti érték lehet. Ezután kiírunk egy szöveget. A **.font**-tal megadjuk a méretét és a stílusát, majd a **.fillText**-el kiírjuk. A **.fillText**-ben adjuk meg magát a szöveget és az x és az y koordinátát. A **.strokeText** kitöltés nélküli szöveget ír ki, a **.drawImage** pedig a korábban készített **Image** object-et teszi ki.

```
<html>
   <head>
      <style type="text/css">
         canvas { border: 1px solid black; }
      </style>
   </head>
   <body onload="draw();">
      <canvas id="canvas1" width="300" height="150"></canvas>
      <script>
         // ennek a 2 sornak még a draw() function előtt le kell futnia
         var img = new Image();
         img.src = "potty.png";
         function draw() {
            var canvas = document.getElementById('canvas1');
            var ctx = canvas.getContext('2d');
            // piros téglalap
            ctx.fillStyle = "rgb(200, 0, 0)";
            ctx.fillRect (10, 10, 55, 50);
            // félig átlátszó kék téglalap
            ctx.fillStyle = "rgba(0, 0, 200, 0.5)"; // 0.5: transparency
            ctx.fillRect (30, 30, 55, 50);
            // kék szöveg, mert a ctx.fillStyle még mindig él
            ctx.font="30px Arial";
            ctx.fillText("Hello", 100, 50);
            // üres szöveg
            ctx.strokeText("Hello", 100, 100);
            // kép kirajzolása
            ctx.drawImage(img, 10, 120);
         }
      </script>
   </body>
</html>
```

A következő példában egy négyzetet mozgatunk jobbra a canvas-on.

A **step** (magyarul: lépés) nevű változóban számoljuk, hogy hányszor futott le a render function. Ha még nem futott le 30-szor, akkor a program alján a **setTimeout** segítségével újra lefuttatjuk. Vagyis ez egy rekurzív program lesz.

Minden egyes lépésben, vagyis a **render** function minden egyes futásakor töröljük a **canvas**-t és újra kirajzoljuk a téglalapot. A téglalap bal felső sarkának x koordinátája **step * 10**, és mivel a

step-et mindig eggyel növeljük, a téglalap bal felső sarkának x koordinátája 10-esével fog növekedni.

```
<canvas id="canvas1" width="350" height="70" style="border: 1px solid
black;"></canvas>
<script>
   var canvas = document.getElementById('canvas1');
   var context = canvas.getContext('2d');
   var step = 0;    // számláló
   function render () {
       // canvas töröl
       context.clearRect(0, 0, canvas.width, canvas.height);
       // piros színkitöltés
       context.fillStyle = "rgb(200, 0, 0)";
       // téglalap kirajzol, paraméterek:
       // - canvas-on belül a bal felső sarok x, y koordinátái
       // - téglalap szélessége, magassága
       context.fillRect(step * 10, 10, 50, 50);
       // ha a step nevű számláló még nem ért 30-ig,
       // egy tized másodperc múlva újra meghívjuk a render function-t,
       // és növeljük a számlálót.
       if (step < 30) {
           setTimeout(render, 100);
       }
       step++;
   }
   render();
</script>
```

A következő program egy olyan képet tartalmaz: pottyok.png, ami különböző mozgás fázisokat tartalmaz egymás mellett. Ez is letölthető innen: http://szilagyidonat.hu/tananyag.html.

Ebből egy animációt, vagyis mozgóképet fogunk készíteni.

A pottyok.png-n belül az 5 kis kép bal felső sarkának x koordinátái: 0, 20, 40, 60, 80. A **setInterval** segítségével egy tized másodpercenként meghívjuk a render function-t. Eközben a step változó értékét 0-tól 4-ig növeljük, majd ha 4-ig ért újra 0-tól 4-ig növeljük. És így tovább. Ezért a kirajzolásnál, amikor meghatározzuk a kis kép bal felső sarkának x koordinátáját, **step * 20** -at fogunk megadni.

```
<canvas id="canvas1" width="100" height="100" style="border: 1px solid black;">
</canvas>
<script>
   var canvas = document.getElementById('canvas1');
   var context = canvas.getContext('2d');
   var img = new Image();
   img.src = "pottyok.png";
   var step = 0;
   function render() {
```

```
        // canvas töröl, a paraméterek:
        // - bal felső sarok koordinátái
        // - a törlés szélessége és magassága
        context.clearRect(0, 0, canvas.width, canvas.height);
        // kép kirajzol, paraméterek:
        // - az Image object
        // - a képen belül a kirajzolandó négyzet bal felső sarkának koordinátái
        // - a képen belül a kirajzolandó négyzet szélessége, magassága
        // - a canvas-on belül kirajzolás helyének bal felső sarka
        // - a canvas-on belül kirajzolás helyének szélessége, magassága
        context.drawImage(img, step * 20, 0, 20, 16, 40, 42, 20, 16);
        // ha elértünk az utolsó step-ig, újraindulunk a nulladiktól
        if (step >= 4) {
            step = 0;
        }
        // step növel
        step++;
    }
    setInterval(render, 100);
</script>
```

9 FORM

FORM ELLENŐRZÉSE

Egy **form**-on (magyarul: űrlap) található e-mail-t tartalmazó input mező, egy számot tartalmazó input mező és 3 radio gomb kitöltöttségének ellenőrzése.

A **<form>** elemen belül az **<input>** elemekkel kérhetünk be adatokat a felhasználótól. A **<form>** és az **<input>** elemnek is lehet neve, amit **name** tulajdonságban adhatunk meg. Erre a névre akkor van szükségünk, ha JavaScript-be be akarjuk olvasni azt, amit a felhasználó beírt az **<input>** elemekbe.

A form-ra egy **submit** és **reset** gombot is tehetünk. A **submit** gomb meghívja azt a function-t, amit a **<form>** elem **onSubmit** tulajdonságában megadunk. Ebben a function-ben **true** értéket adunk vissza, ha elfogadjuk az input mezők értékeit, és **false** értéket ha hibásnak találjuk őket. A true törli is az input mezők tartalmát. A **reset** gomb csak simán törli a mezőket, nem hív meg function-t.

A részletes leírás a programban található kommentekben van.

```
<html>
<meta charset="utf-8"/>
<body>
<!-- ez HTML komment -->
<!-- ha megnyomjuk a type="submit" gombot, meghívja az ellenoriz() function-t -->
<!-- a type="reset" gomb törli a form-ot -->
<form name="urlap" onSubmit="return ellenoriz();">
    <input type="radio" name="termek"> A termék <br>
    <input type="radio" name="termek"> B termék <br>
    <input type="radio" name="termek"> C termék <br>
    Mennyiség <input type="text" name="meny"> <br>
```

```
    E-mail: <input type="text" name="email"> <br><br>
    <input type="submit" value="Oké">
    <input type="reset" value="Mégse">
</form>
<script>
    // ezt hívja meg a submit gomb
    function ellenoriz() {
        var jo = true;    // itt tároljuk az ellenőrzés végeredményét
        var hiba = '';    // ide gyűjtjük a hibák szövegeit
        // az form-ra a document.urlap segítségével hivatkozunk,
        // mert fenn HTML-ben a form-nak azt a nevet adtuk, hogy "urlap"
        // ha nincs kitöltve a mennyiség,
        // akkor az input mező .value (magyarul: érték) tulajdonsága üres szöveg
        // a "meny" az egyik input mező neve a form-on belül
        if (document.urlap.meny.value == '') {
            hiba += 'Nincs a mennyiség megadva.\n';
            jo = false;
        }
        // ha a mennyiség nem szám, az hiba
        if (isNaN(document.urlap.meny.value)) {
            hiba += 'A mennyiség hibás\n';
            jo = false;
        }
        // ha az input mező értéke nem tartalmaz @ betűt, az hiba
        // az .indexOf egy szövegben keres egy másik szöveget
        // ha a megtalálja, akkor a szöveg pozícióját adja vissza,
        // ha nem, akkor -1 -et
        if (document.urlap.email.value.indexOf('@') == -1) {
            hiba += 'Hibás e-mail.\n';
            jo = false;
        }
        // itt a 3 radio gombot ellenőrizzük
        var termekvalasztva = false;    // ide gyűjtjük, hogy van-e radio kiválasztva
        // itt végigmegyünk a 3 radio gombon
        // mivel hárman is vannak azonos névvel, a document.urlap.termek array lesz
        // a .checked tulajdonság egy boolean, true ha be van pipálva a radio
        for (var v = 0; v < 3; v++) {
            if (document.urlap.termek[v].checked) {
                termekvalasztva = true;
            }
        }
        if (!termekvalasztva) {
            hiba += 'Nincs termék kiválasztva.\n';
            jo = false;
        }
        if (!jo) {
            // ha nem jó a form, akkor kiírjuk a hibát
            window.alert('Hiba!\n' + hiba);
            // az onSubmit által meghívott function false-t ad vissza, ha hiba van
            return false;
        } else {
```

```
            window.alert('Nincs hiba.');
            // az onSubmit által meghívott function true-t ad vissza, ha nincs hiba
            return true;
        }
    }
</script>
</body>
</html>
```

ÖSSZEAD

A következő program a "Számol" gomb megnyomásra két szövegmezőbe beírt számot összead és egy harmadik szövegmezőben megjeleníti az eredményt. Ha nem számot adunk meg, egy alert ablakban figyelmeztet.

A **size** az input mezők szélességét állítja be. Az **input type="button"** egy sima gomb (se nem submit se nem reset), amin van egy **onclick** eseménykezelő. A **Number()** számmá alakít egy szöveget, amire azért van szükség, hogy ne szövegként, hanem számként adjuk össze a két mező értéket. Az **isNaN**-t és a **VAGY** műveletet **||** már ismerjük.

```
<meta charset="utf-8"/>
<form name="urlap">
    <input name="mezo1" size="2"> +
    <input name="mezo2" size="2"> =
    <input name="eredm" size="2"> <br><br>
    <input type="button" value="Számol" onclick="osszead();">
</form>
<script>
function osszead() {
    var a = Number(document.urlap.mezo1.value);
    var b = Number(document.urlap.mezo2.value);
    if (isNaN(a) || isNaN(b)) {
        alert("Nem számot írtál be!");
    } else {
        document.urlap.eredm.value = a + b;
    }
}
</script>
```

LINKEK

Linkeket tartalmazó **select** lenyíló menü. Be is tölti a linkeket az **onchange** eseménykezelő hatására. Az **onchange**-ben megadott function akkor hívódik meg, ha kiválasztunk egy **option**-t (magyarul: lehetőség) a **select** lenyílóban. A **document.urlap.menu.selectedIndex** a kiválasztott **option** sorszámát (angolul: **index**) adja vissza ami most 0, 1 vagy 2 lehet. A **document.urlap.menu[document.urlap.menu.selectedIndex].value** visszaadja a kiválasztott option **value** (magyarul: érték) tulajdonságát. A **window.location** irányítja át a böngészőt az új weboldalra.

```
<form name="urlap">
    <select name=menu onchange="ugras();">
        <option value="http://idokep.hu">IDOKEP</option>
        <option value="http://koponyeg.hu">KOPONYEG</option>
```

```
        <option value="http://met.hu">MET</option>
    </select>
</form>
<script>
function ugras() {
    window.location = document.urlap.menu[document.urlap.menu.selectedIndex].value;
    return true;
}
</script>
```

CHECK ALL

Négy **checkbox**-ot jelenítünk meg, ha a negyediket beikszelem, az első három is beikszelődik. A negyedik checkbox **onclick** eseménykezelője meghívja a checkall nevű function-t. A checkall a **.checked** segítségével ellenőrzi, hogy a negyedik checkbox tényleg ki van-e pipálva. Ha igen, akkor egy **for** ciklussal végigmegy a felső három checkbox-on és a **.checked** segítségével mindet kipipálja.

```
<form name="myform">
    <input type="checkbox" name="box" value="1"> 1 <br>
    <input type="checkbox" name="box" value="2"> 2 <br>
    <input type="checkbox" name="box" value="3"> 3 <br>
    <input type="checkbox" name="all" onclick="return checkall();"> mind <br>
</form>
<script>
function checkall() {
    if (document.myform.all.checked) {
        for (var i = 0; i < 3; i++) {
            document.myform.box[i].checked = true;
        }
    }
    return true;
}
</script>
```

NEM ÜRES MEZŐ

A következő példa olyan szövegmezőket jelenít meg egy form-on, melyeket nem lehet üresen hagyni, mert ha kitöröljük az értéküket, egy x íródik beléjük. A form submit-álását most nem egy submit gombbal, hanem egy checkbox-ra történő klikkelessel történhet.

Az **input** elem **.value** tulajdonsága beállítja a mezőbe írt értéket.

Az **.onblur** eseménykezelő akkor hívja meg a mögötte lévő programot, ha ellépünk a szövegmezőből, például úgy, hogy egy másik szövegmezőre klikkelünk. Az eseménykezelőben a **this** az **event.currentTarget**-nek felel meg, vagyis azt az elemet adja vissza, amire az eseménykezelőt tettük. A lenti példában ha ennek az elemnek a **.value** tulajdonsága üres szöveg, akkor átírjuk x-re. Az **.onblur** ellentéte az **.onfocus**, ami akkor aktiválódik, ha kiválasztunk egy mezőt.

A **this.form** visszaadja azt a form-ot, amiben vagyunk. A **.submit()** pedig submit-olja a form-ot.

```
<meta charset="utf-8"/>
<form>
```

```
    Ezeket a mezőket nem lehet üresen hagyni: <br>
    <input type="textbox" size="5" value="t1"
        onblur="if (this.value=='') { this.value='x'; }"> <br>
    <input type="textbox" size="5" value="t2"
        onblur="if (this.value=='') { this.value='x'; }"> <br>
    <br>
    Input elem, melyre klikkelve az elküldi magát: <br>
    <input type="checkbox" onClick="this.form.submit();"> <br>
</form>
```

CSAK EGY SUBMIT

A következő példában a form submit gombjának lenyomására egy function **disabled**-re állítja a submit gombot, majd betölti a success.html oldalt. A **disabled**-re állítás elszürkíti a gombot és nem engedi, hogy megnyomjuk.

Az **onsubmit** eseménykezelő azelőtt hívódik meg, hogy a form submit-olna az **action**-ben megadott success.html oldalra. A form adatait ezután elvileg a success.html dolgozná fel. Az **action** a success.html helyett a való életben egy szerver oldali programra szokott mutatni, ami nem a böngészőben fut, és jó eséllyel más programnyelven íródott.

```
<html>
<meta charset="utf-8"/>
<body>
    <form name="myform" onsubmit="disableSubmit()" action="success.html">
        <input type="text" value="abc"> <br>
        <input type="submit" name="mysubmit" value="Submit">
    </form>
    <script>
        function disableSubmit() {
            document.myform.mysubmit.disabled = true;
        }
    </script>
</body>
</html>
```

A success.html oldal így néz ki:

```
<B>Success!</B>
```

Ha egy form-ot submit-tálunk, érdemes arról gondoskodni, hogy ne tudja kétszer megnyomni a submit gombot a felhasználó.

10 FÁJL BEOLVASÁSA

Csináljunk a **.html** kiterjesztésű fájlunk mellé egy **valami.txt** nevű fájlt, és írjunk bele valami szöveget.

A következő program a **fetch** paranccsal beolvassa a valami.txt fájlt, miután ez megtörtént, a kinyeri a fájl szövegét, és miután ez megtörtént a szöveget beleteszi a **div1** elembe.

A **.then**-ekre azért van szükség, mert a **fetch** nem várja meg, hogy a fájl betöltődjön, azonnal továbblép a következő programsorra. A **.then** viszont megvárja és a **fetch** eredményét beteszi egy **Response** object-be, amit a => után használhatunk. Ennek a **Response** object-nek a

szöveges tartalmát a **.text()** segítségével nyerjük ki. Ezután a **.then** is beteszi az eredményét egy változóba, amit a következő **.then** fog kiolvasni. A lenti példában a **result** változóba fog kerülni a **response.text()** értéke.

A **fetch** az eredményt egy **Response** object-be teszi bele, aminek a szöveges tartalmát a **.text()** segítségével nyerjük ki.

```
<meta charset="utf-8"/>
<div id="div1"></div>
<script>
    fetch('valami.txt')
        .then(response => response.text())
        .then(result => document.getElementById('div1').innerHTML = result);
</script>
```

Ha a **script** elemben még lenne ezután valami programrész, az valószínűleg hamarabb futna le, mint a **.then**-ek. Ezt a módszert, amikor egy programrész csak később fog lefutni **aszinkron** programozásnak hívják, és hasonlóan működik, mint a korábban tanult **window.setTimeout()**.

11 DEBUG

MI AZ A DEBUG?

A **bug** magyarul bogarat vagy hibát jelent a programban. A **debug** az, amikor a hibákat megkeressük, majd kijavítjuk a programban.

A **debug** egy olyan program futtatási módot is jelent, amikor a program futását bármikor megállíthatjuk és megnézhetjük a változók értékét, majd folytathatjuk a program futtatását. Az összes böngésző támogatja, hogy a JavaScript programunkat debug módban futtassuk. Mi a Firefox és a Chrome böngészőkben fogjuk ezt megnézni, de a többi böngészőben is hasonlóan működik.

PÉLDAPROGRAM, AMIT DEBUG-OLNI FOGUNK

Debug üzemmódban néhány böngésző elvárja, hogy a HTML és a JavaScript külön fájlban legyen. De a profik amúgy is külön fájlba szokták tenni őket. Ezért mi is egy ilyen példán fogunk gyakorolni. Szóval csináljunk két fájlt: 1.html és 1.js.

1.html

```
<html><body>
<div id="div0"></div>
<div id="div1"></div>
<div id="div2"></div>
<script src="1.js"></script>
</body></html>
```

1.js

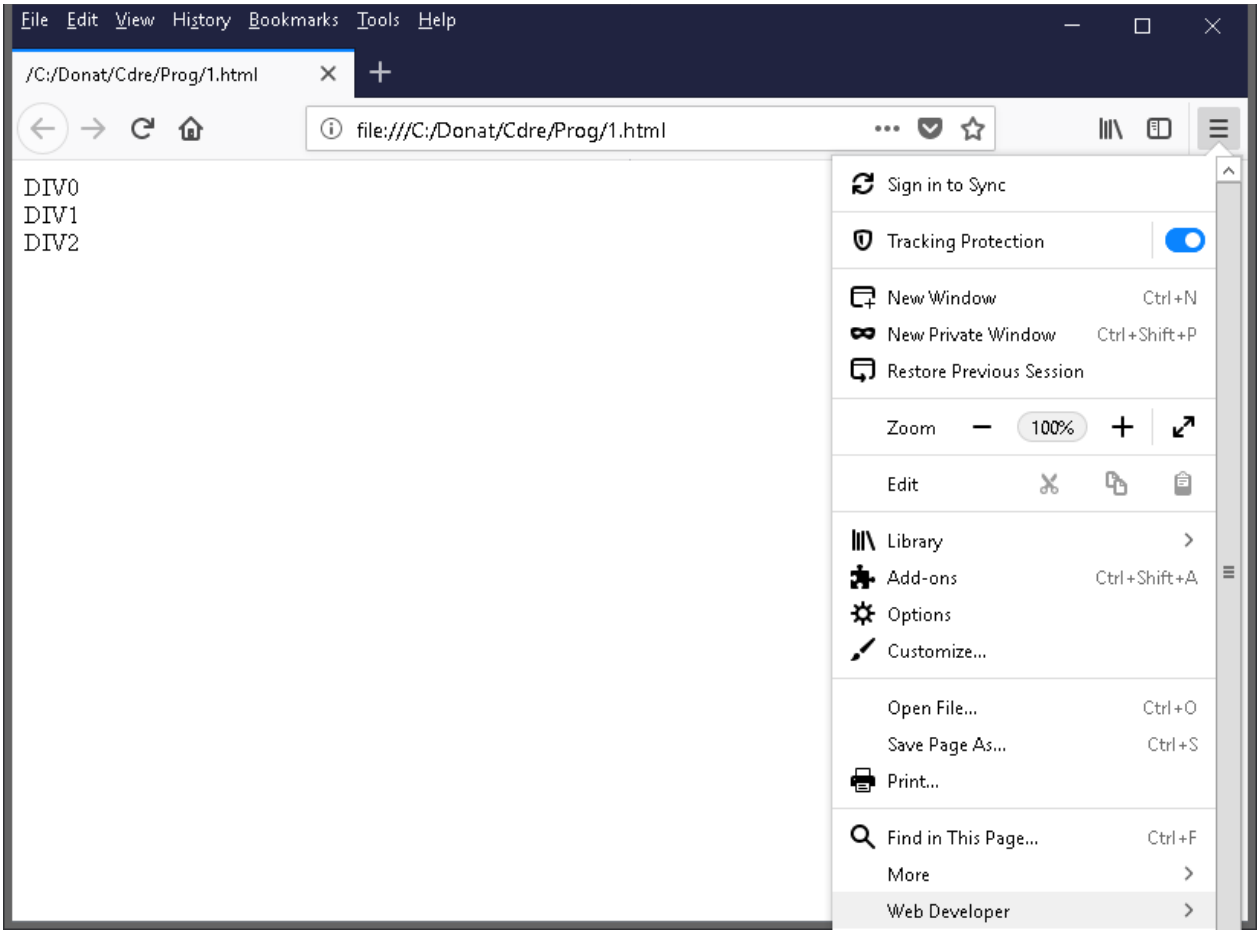
```
var v = 'DIV';
for (var i = 0; i < 3; i++) {
    document.querySelector('#div' + i).innerHTML = v + i;
}
```

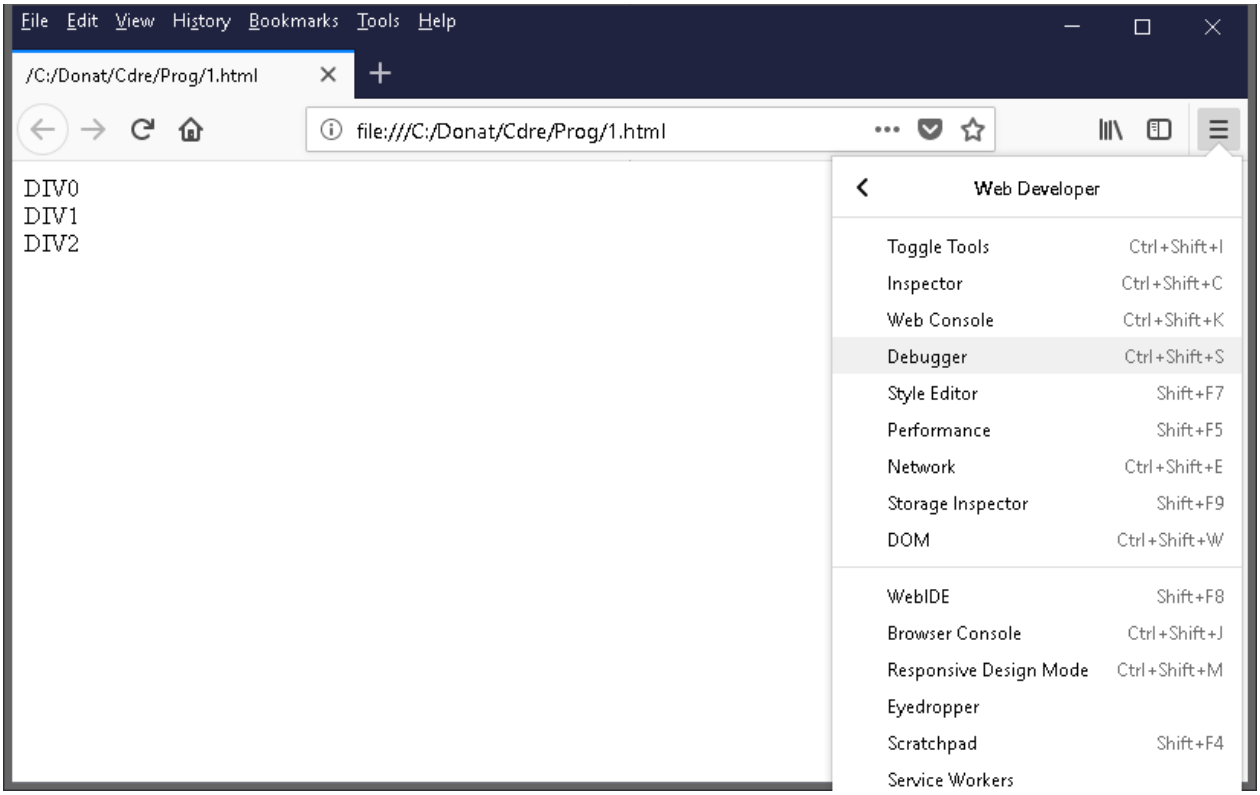
DEBUG FIREFOX-SZAL

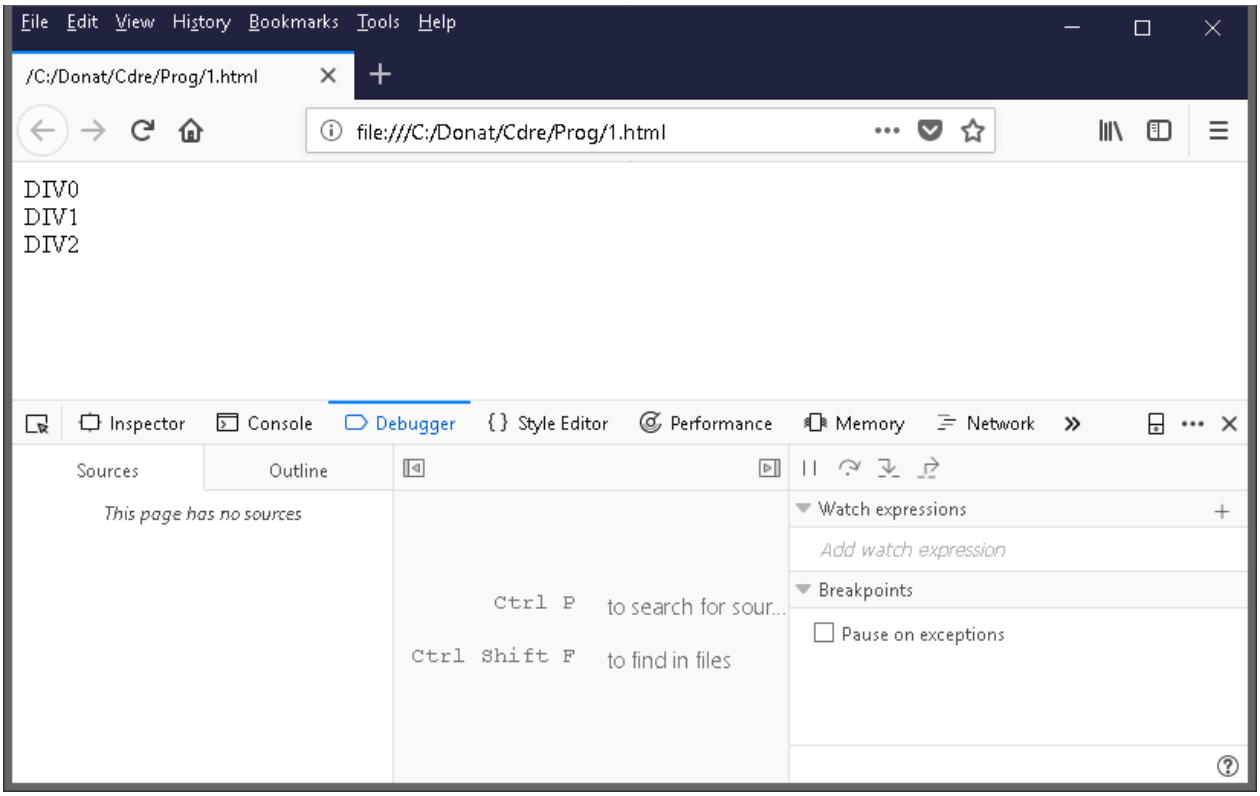
Futtassuk az 1.html-t Firefox böngészővel.

Indítsuk el a Firefox debugger-ét.

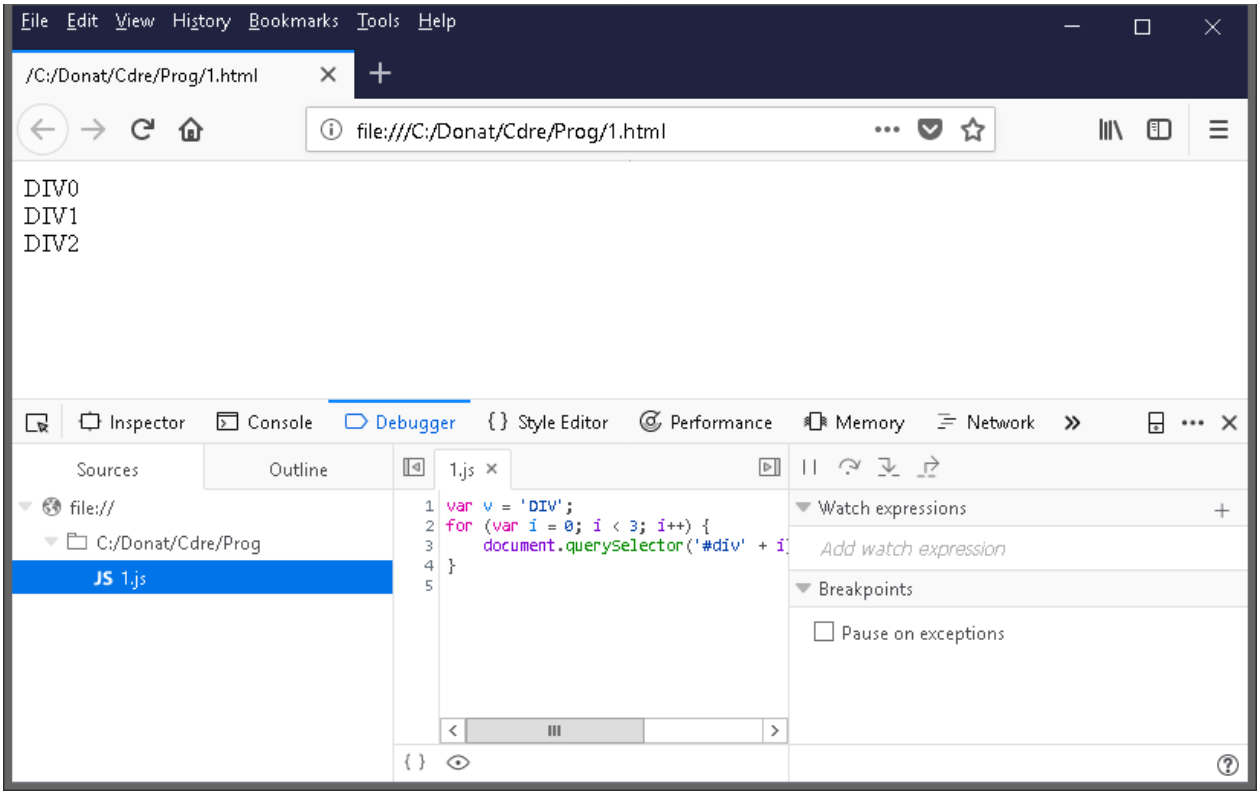
Menu (3 vízszintes vonal a jobb felső sarokban) > Web Developer > Debugger





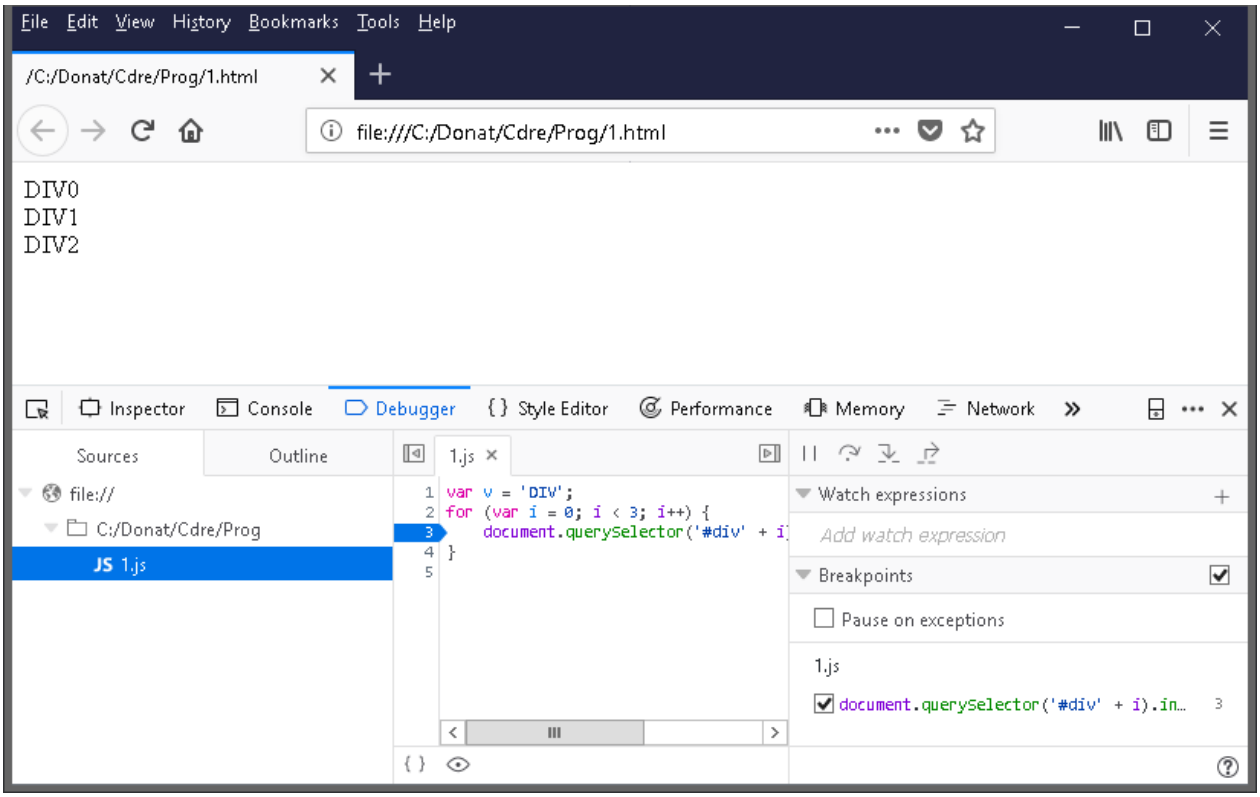


Nyomjunk egy F5 gombot, majd a Sources fül alatt keressük meg az 1.js fájlt és klikkeljünk rá.



Középen láthatjuk a programunkat az 1.js fül alatt.

Csináljunk egy **breakpoint**-ot. A breakpoint egy olyan hely a programban, ahol a program futását meg fogjuk állítani. Középen az 1.js fül alatt a program bal szélén klikkeljünk az egyik sorszámra.

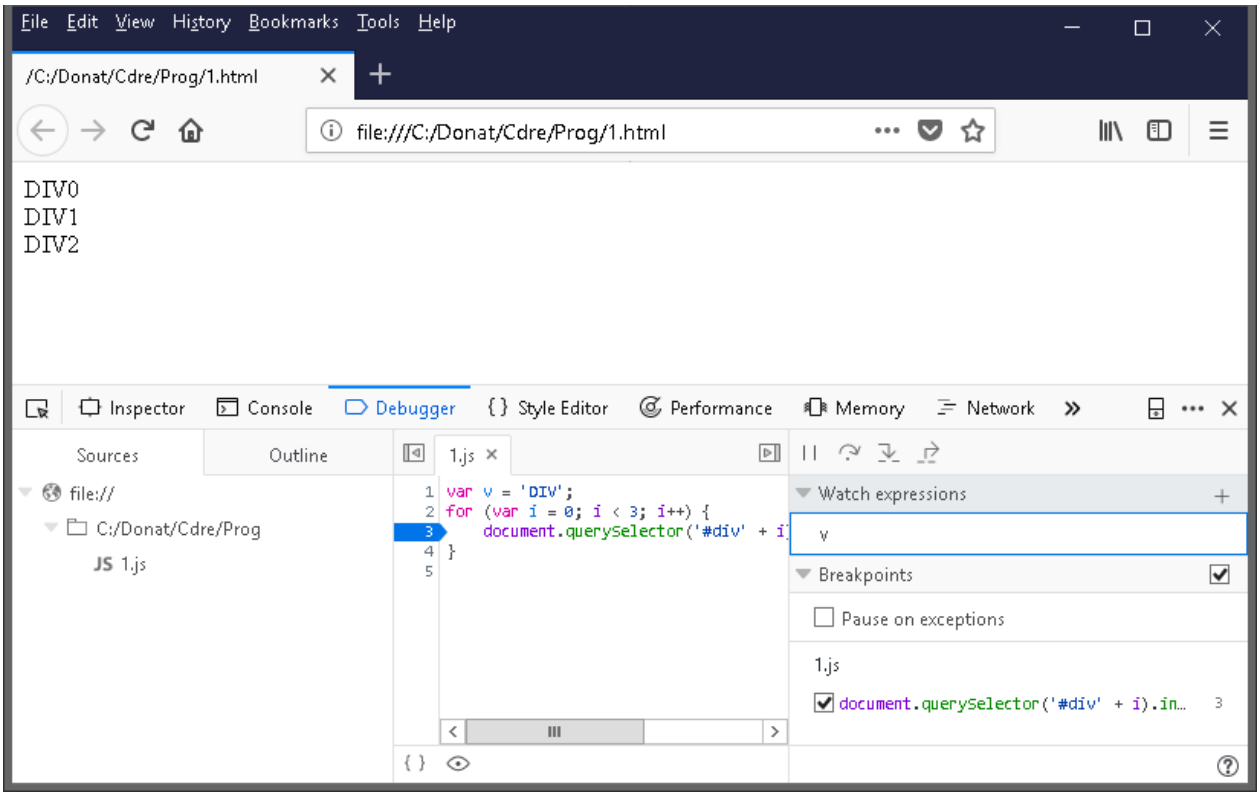


Ekkor a jobb oldalon a **Breakponts** fül alatt is megjelenik a breakpoint-unk.

Ha el akarjuk tűntetni a breakpoint-ot, újra a sorszámra kell klikkelni, de most hagyjunk legalább egy breakpoint-ot bekapcsolva.

Watch expression beállításával tudjuk egy változó értékét vagy egy tetszőleges kifejezés értékét megfigyelni.

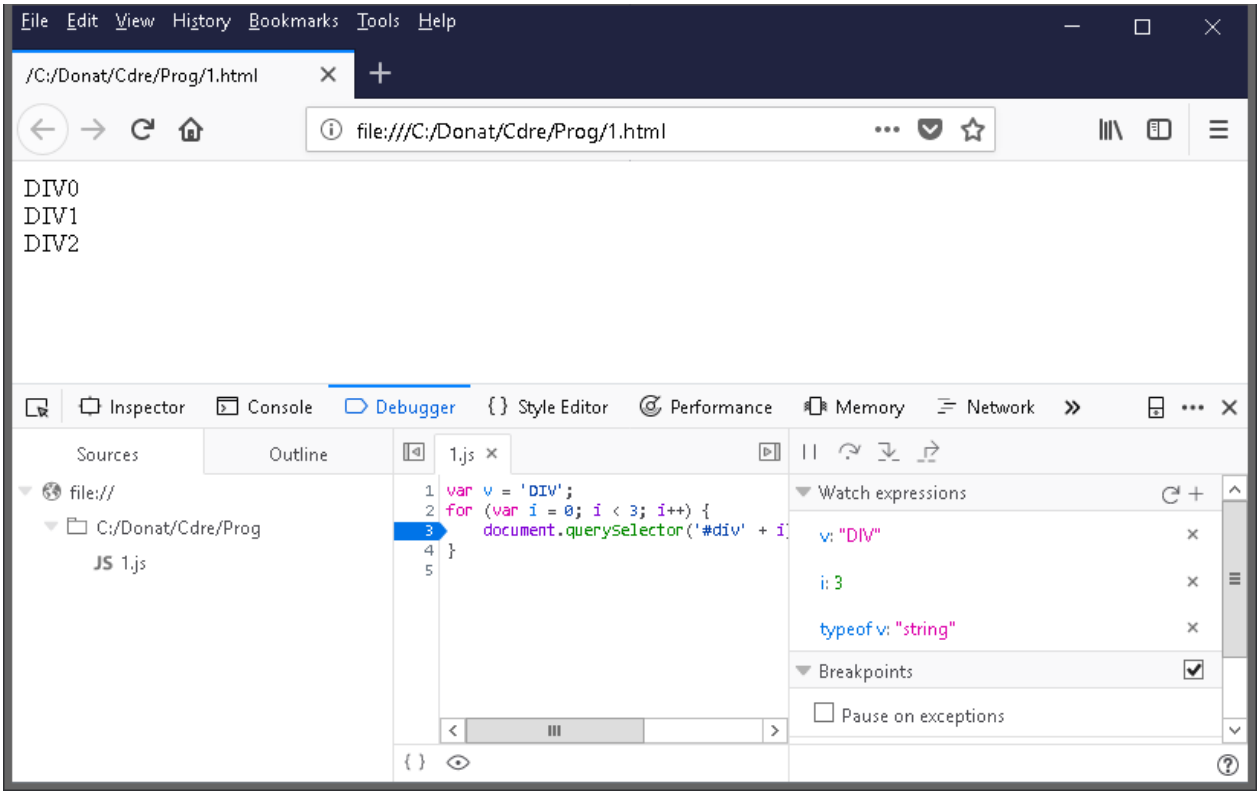
Jobboldalt a **Watch expressions** mellett a + gombbal tudjuk beírni, hogy mit akarunk figyelni.



Írjuk be az alábbi 3 dolgot:

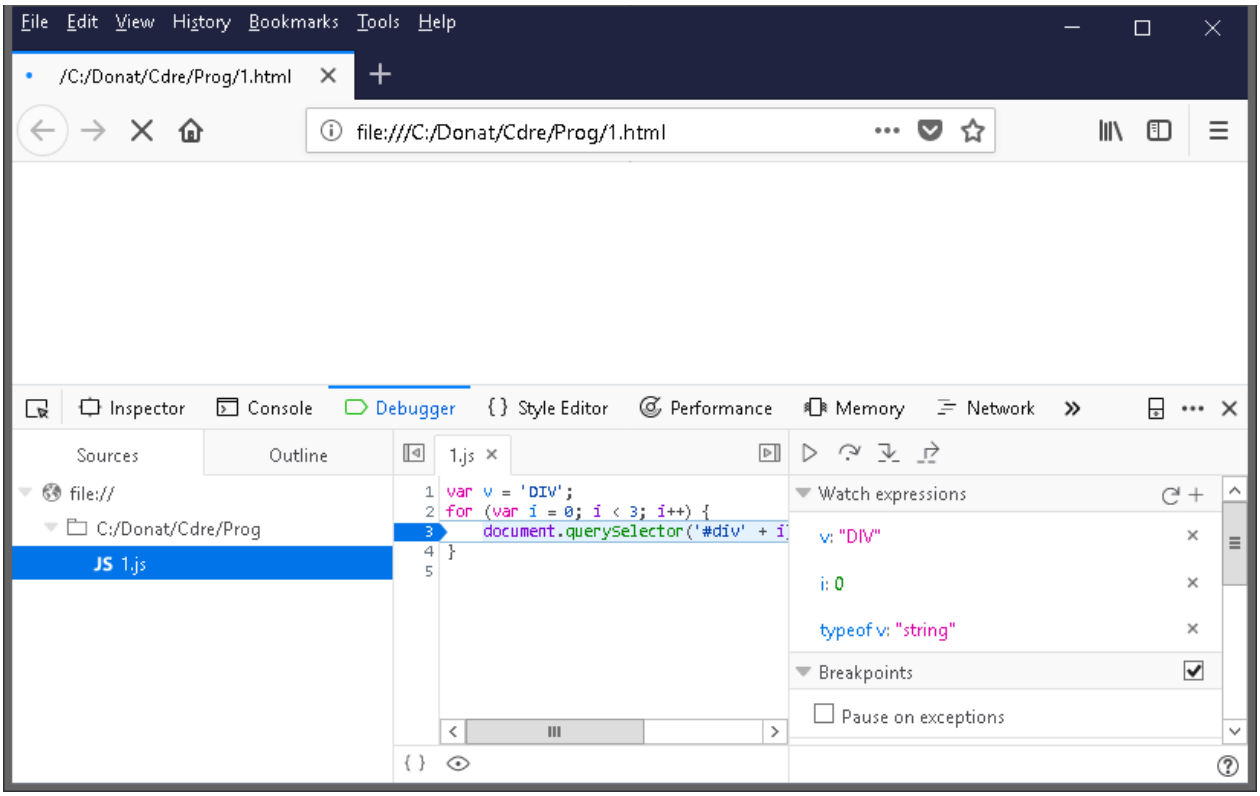
v
i
typeof v

Minden sor után enter-t kell nyomni.



A sor bal oldalán lévő **x** segítségével tudunk törölni.

Itt az ideje, hogy elkezdjünk debug-olni. Nyomjunk F5 gombot, hogy újraindítsuk a programunkat. A program meg fog állni az első break point-nál.



A Watch expressions fül alatt láthatjuk azoknak a kifejezéseknek az értékeit, amiket megadtuk.

A Watch expressions fül fölött keressük meg az alábbi ikonokat:

Ezeket akkor használhatjuk, amikor a debugger megáll egy breakpoint-on. Jelentésük balról jobbra:

- Resume: a program fusson a következő breakpoint-ig

- Step over: lépjen a következő sorra ugyanebben a function-ben

- Step in: ha a következő sor egy function-t hív meg, akkor menjen bele, ha nem akkor lépjen a következő sorra

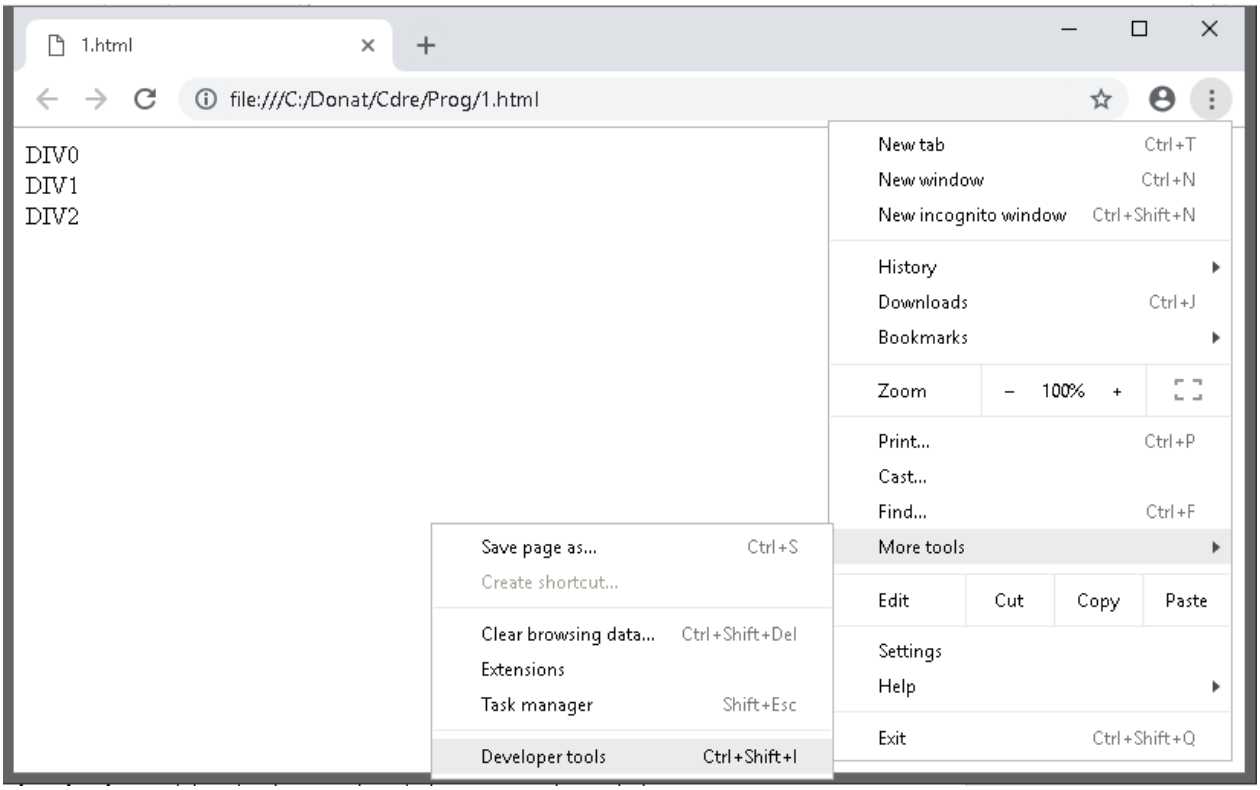
- Step out: fusson annak a function-nek a végéig, amelyikben vagyunk

DEBUG CHROME-MAL

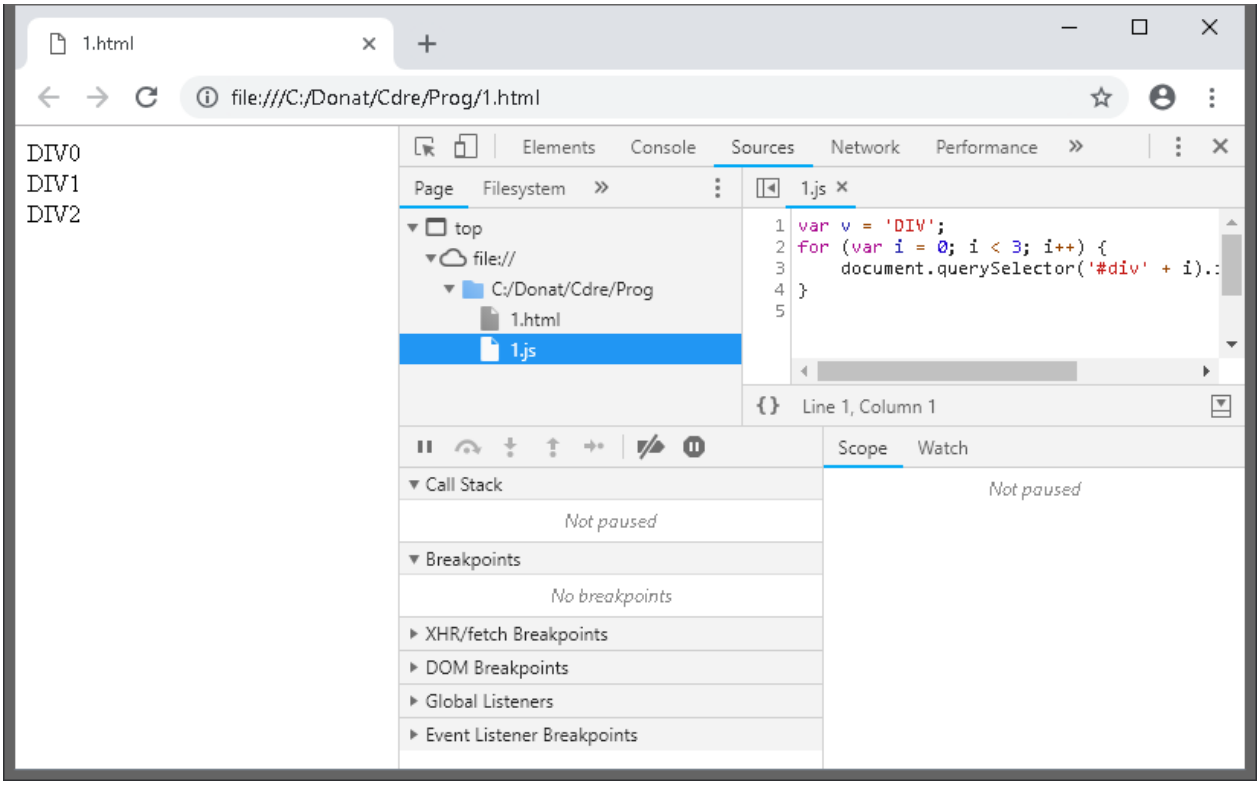
Futtassuk az 1.html-t Chrome böngészővel.

Indítsuk el a Chrome debugger-ét.

Menu (3 pont a jobb felső sarokban) > More tools > Developer tools

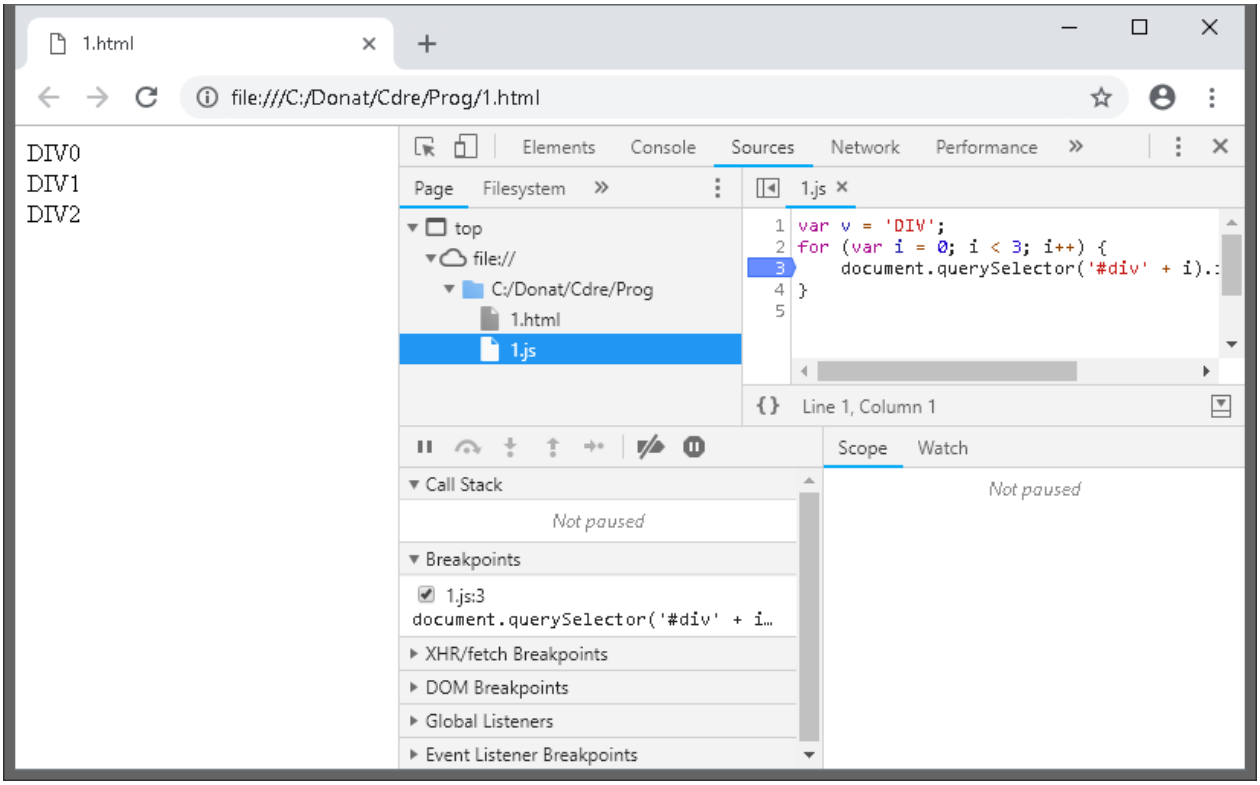


Nyomjunk egy F5 gombot, majd a Sources fül alatt keressük meg az 1.js fájlt és klikkeljünk rá.



A jobb oldalon láthatjuk a programunkat az 1.js fül alatt.

Csináljunk egy **breakpoint**-ot. A breakpoint egy olyan hely a programban, ahol a program futását meg fogjuk állítani. Középen az 1.js fül alatt a program bal szélén klikkeljünk az egyik sorszámra.

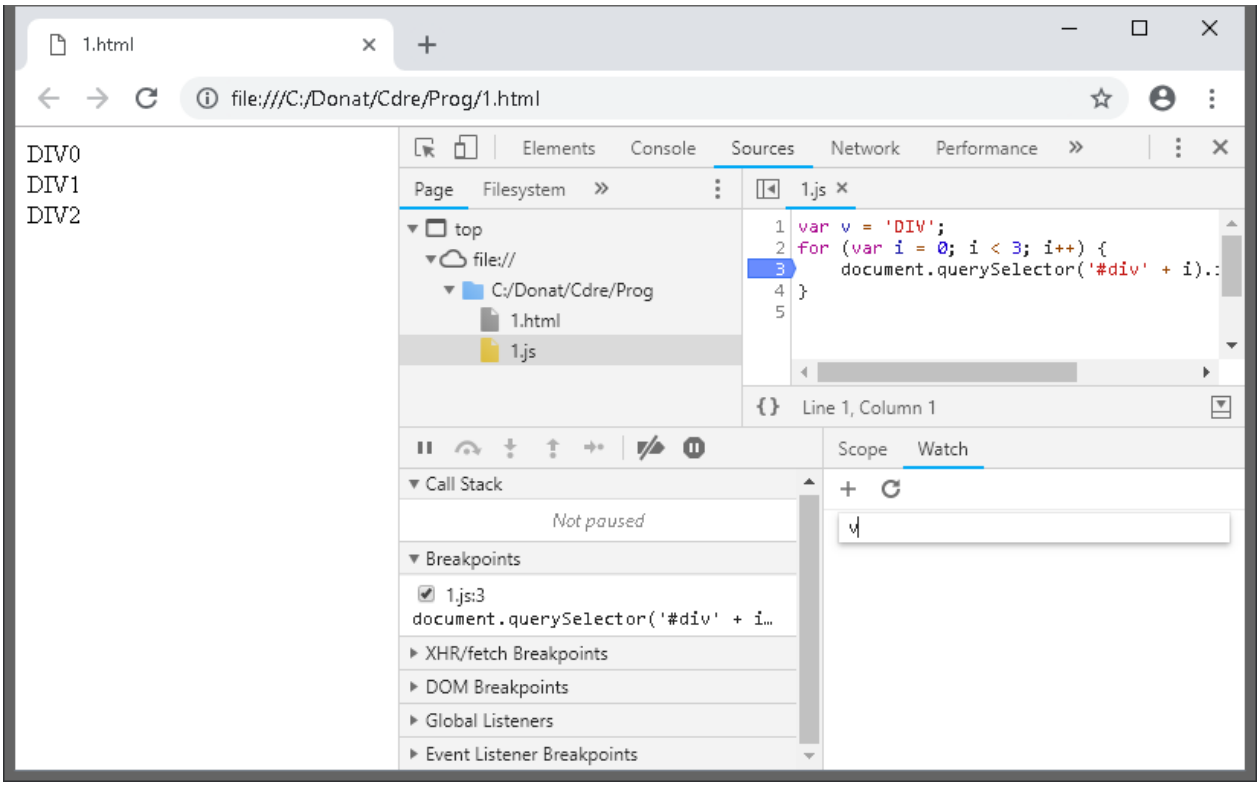


Ekkor középen alul a **Breakponts** fül alatt is megjelenik a breakpoint-unk.

Ha el akarjuk tűntetni a breakpoint-ot, újra a sorszámra kell klikkelni, de most hagyjunk legalább egy breakpoint-ot bekapcsolva.

Watch beállításával tudjuk egy változó értékét vagy egy tetszőleges kifejezés értékét megfigyelni.

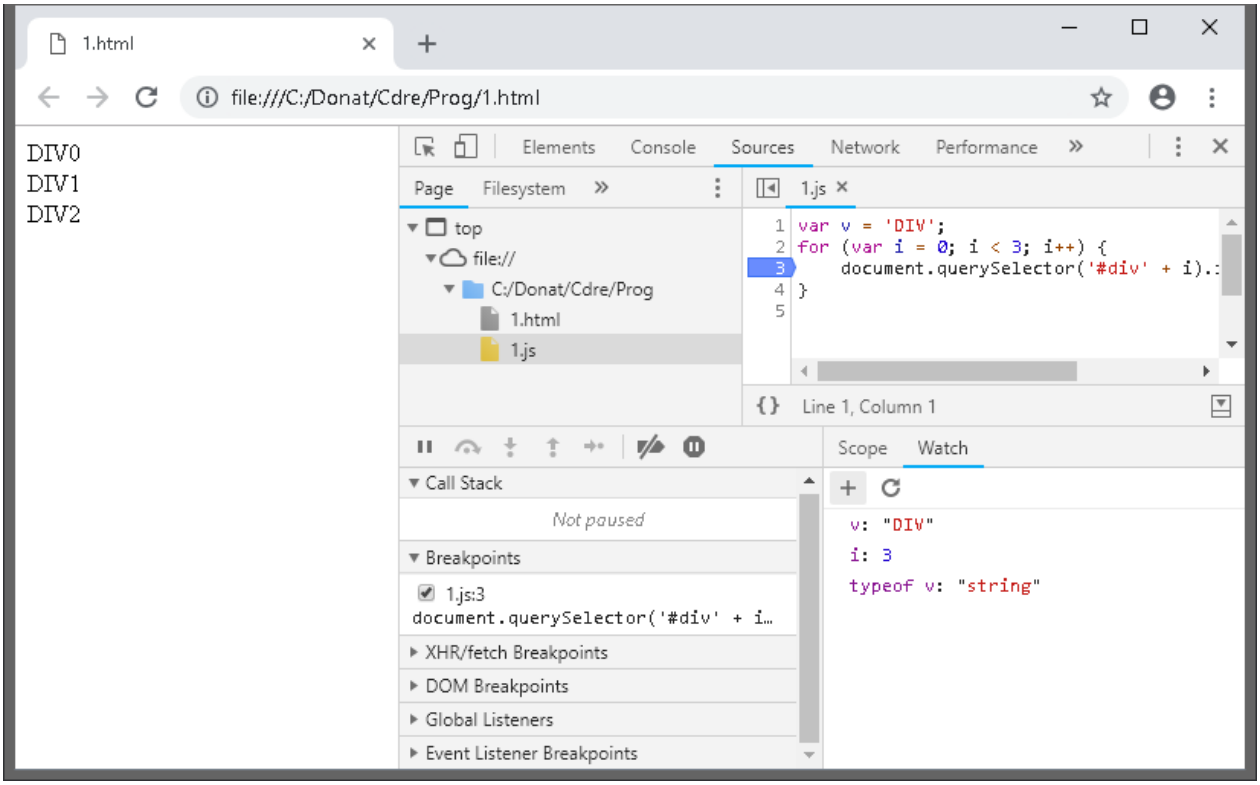
Jobboldalt a **Watch** alatt a + gombbal tudjuk beírni, hogy mit akarunk figyelni.



Írjuk be az alábbi 3 dolgot:

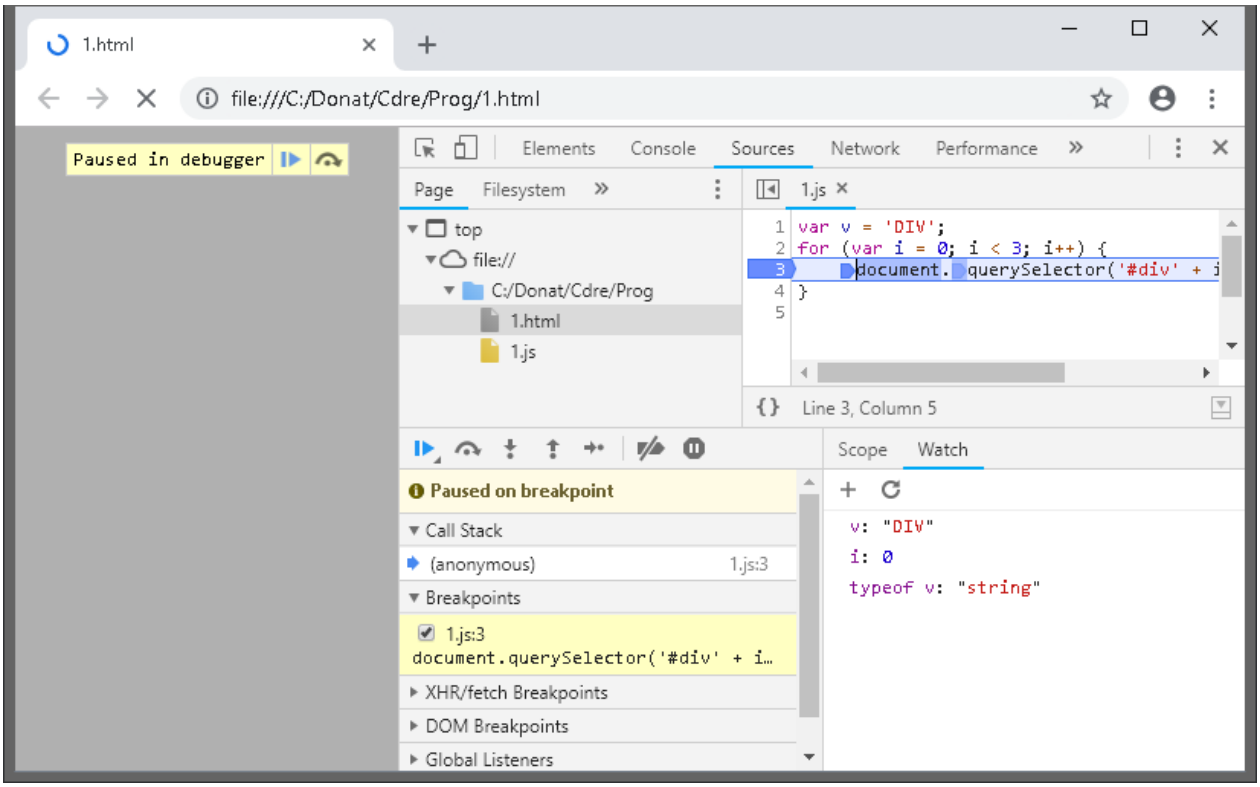
v
i
typeof v

Minden sor után kétszer enter-t kell nyomni.



Ha az egyik sor fölé megyünk egérrel, a bal oldalán megjelenik egy – ikon, ezzel tudunk törölni.

Itt az ideje, hogy elkezdjünk debug-olni. Nyomjunk F5 gombot, hogy újraindítsuk a programunkat. A program meg fog állni az első break point-nál.



A Watch fül alatt láthatjuk azoknak a kifejezéseknek az értékeit, amiket megadtuk.

Középen keressük meg az alábbi ikonokat:

Ezeket akkor használhatjuk, amikor a debugger megáll egy breakpoint-on. Jelentésük balról jobbra:

- Resume script execution: a program fusson a következő breakpoint-ig

- Step over next function call: lépjen a következő sorra ugyanebben a function-ben

- Step into next function call: ha a következő sor egy function-t hív meg, akkor menjen bele, ha nem akkor lépjen a következő sorra

- Step out of current function: fusson annak a function-nek a végéig, amelyikben vagyunk

- Step: lépjen egy sorral tovább

VÉGE