Belépő a tudás közösségébe

Informatika szakköri segédanyag

A böngésző mint alkalmazásfejlesztési platform

Horváth Győző, Visnovitz Márton

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban” (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2017-ben.

BEFEKTETÉS A JÖVŐBE

A BÖNGÉSZŐ MINT ALKALMAZÁSFEJLESZTÉSI PLATFORM

Horváth Győző, Visnovitz Márton

Belépő a tudás közösségébe

Informatika szakköri segédanyag

A kiadvány „A felsőoktatásba bekerülést elősegítő készségfejlesztő és kommunikációs programok megvalósítása, valamint az MTMI szakok népszerűsítése a felsőoktatásban" (EFOP-3.4.4-16-2017-006) című pályázat keretében készült 2017-ben.

ISBN 978-963-284-993-5

TARTALOMJEGYZÉK

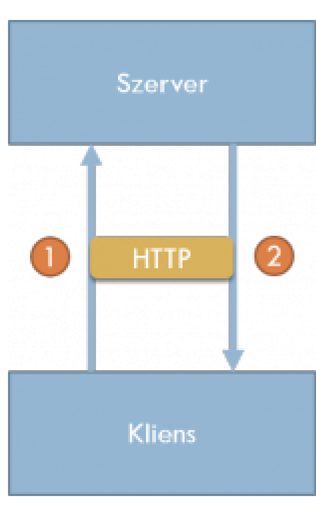
1. **Bevezetés ↪**
2. **Kliensoldali dinamikus webprogramozás és eszközei ↪**
3. **A JavaScript programozási nyelv ↪**
4. **Felületi elemek programozása ↪**
5. **Interaktív programok – eseménykezelés ↪**
6. **A kliensoldali alkalmazásfejlesztés alapelvei ↪**
7. **Rajzolás JavaScript segítségével ↪**
8. **Szimulációk és játékok készítése vászonnal ↪**
9. **Adatok mentése a böngészőben ↪**

1 BEVEZETÉS

1.1 Dinamikus weboldalak

Manapság számítógépes tevékenységeink tekintélyes részét a böngészőprogram használata jelenti. Információkat keresünk, híreket olvasunk, videókat nézünk, kapcsolatot teremtünk az ismerőseinkkel, egyre több hivatalos ügyet el tudunk intézni online, egyszerűbb játékokat is játszhatunk. Mindezeket valamilyen webes alkalmazás segítségével tudjuk megtenni, így joggal mondhatjuk, hogy a böngészők a webes technológiákkal együtt modern alkalmazásfejlesztési platformmá nőtték ki magukat.

A webes alkalmazások – a web jellegéből fakadóan – *kliens-szerver architektúrában* működnek. A szerver közzéteszi az elérhető erőforrásokat (HTML dokumentumok, képek, stb.), ezeket pedig klienssel, azaz böngészővel kérhetjük el a szervertől. *Dinamikus weboldalakról* akkor beszélünk, ha a megjelenített dokumentum előállításához, működtetéséhez, módosításához valamilyen számítógépes programot használunk. Ez a program futhat szerveroldalon, ekkor a böngészőnek leküldendő tartalmat dinamikusan állítjuk elő ezzel a programmal; vagy futhat kliensoldalon, ekkor a böngészőbe már betöltött HTML oldal dinamikus működtetése a cél. Egy összetettebb webalkalmazásban mindkét oldalon használhatunk programot.



1.2 A tananyag célja

Ebben a tananyagban a kliensoldali webfejlesztésre fókuszálunk. Elsősorban azt fogjuk megnézni, hogy *hogyan használható a böngésző grafikus programok készítésére*. A böngészők manapság olyan sokféle szolgáltatást nyújtanak programozási szempontból, hogy méltó alternatívái tudnak lenni az asztali alkalmazásoknak: sokféle grafikus elemet képesek megjeleníteni, ezeket kényelmesen, eseményvezérelt módon tudjuk programozni, jó pár adattárolási lehetőség közül választhatunk. Egy böngészőben futó webes alkalmazás nagy előnye, hogy bármikor könnyen publikálható egy webszerveren, nem kell külön telepíteni, és gyakorlatilag bármilyen operációs rendszeren vagy mobil eszközön elérhető, amelyen van valamilyen korszerű böngészőprogram.

A tananyagban olyan alkalmazások elkészítését tűzzük ki célul, melyekben HTML leíró nyelv segítségével írjuk le a felhasználói felület szerkezetét (a dokumentumot), CSS leíró nyelv segítségével határozzuk meg a kinézetét, és JavaScript programozási nyelv segítségével adjuk hozzá a szükséges viselkedési logikát.

A tananyag elsajátításával a diákok képesek lesznek egyszerűbb, böngészőben futó alkalmazások elkészítésére. Ilyen alkalmazások lehetnek például:

- egyszerű játékok (logikai, grafikus, ügyességi);
- számolási segédeszközök;
- szimulációk (szimulációs adatok számolása és megjelenítése).

A tananyag végére az aknakereső játékot is elkészítjük.

1.3 A tananyag felépítése

A tananyag több, egymásra épülő fejezetet tartalmaz. Minden fejezet elején röviden ismertetjük az adott témakör *elméleti tudnivaló*it, majd azok használatát *több, kisebb feladat*on keresztül mutatjuk be. A tananyagot végigkísérik *témakörökön átívelő nagyobb feladat*ok is, ezeket minden témakörnél az új ismeretek fényében tovább fejlesztjük. A fejezetek végén további *gyakorló feladatok* kapnak helyet.

A tananyagban a következő jelöléseket használjuk:

> **Példa**
>
> A tananyaghoz tartozó gyakorlati feladatokat, szemléltető szöveg- és kódrészleteket tartalmazó blokkba.

> **Ismétlés**
>
> A feltételezett ismeretek gyors áttekintésére szolgáló blokk.

Apróbetűs

A törzsanyagon túlmutató, további részleteket, kapcsolódó érdekességeket bemutató vagy továbblépési lehetőségeket felvillantó ismereteket ilyen blokkban közöljük.

Nagyfeladat

A tananyagot végigkísérő nagyobb feladathoz tartozó részfeladatokat tartalmazzák ezek a blokkok.

Letöltés

A letölthető anyagok ilyen blokkokban jelennek meg.

1.4 Szükséges előismeretek

Habár a tananyag kezdőknek szól, bizonyos mértékben épít korábbi tapasztalatokra, ismeretekre. A feltételezett előismeretek az alábbiak:

- HTML jelölőnyelv ismerete,
- CSS szelektorok és tulajdonságok alapfokú ismerete,
- Alapvető algoritmizálási és programozási ismeretek (programozási tételek).

2 KLIENSOLDALI DINAMIKUS WEBPROGRAMOZÁS ÉS ESZKÖZEI

2.1 A világháló és technológiái

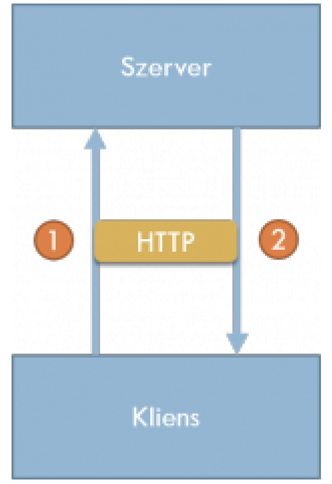
A *világháló* (angolul World Wide Web, röviden web) egy olyan információs rendszer, amelyben dokumentumokat és más erőforrásokat egységes címmel azonosítunk, ezeket hiperhivatkozásokkal kötjük össze, elérhetőségüket pedig internetre kötött szerverek segítségével biztosítjuk. A web több komponensből épül fel, működését számos szabvány, protokoll és technológia biztosítja:

- A weben keresztül elérhető *dokumentumok* (weboldalak) leírása *HTML nyelv* ↪ segítségével történik. Ezek további erőforrásokra (dokumentumok, képek, stb) való *hiperhivatkozások*at tartalmazhatnak.
- Az erőforrásokat *egységes cím*mel azonosítjuk (**URL** ↪), amely tartalmazza, hogy milyen protokollon melyik szerveren hol is van az adott erőforrás (`http://pelda.szerver.hu/ut/cel.html`)
- Az erőforrásokat *webszerver*ek teszik elérhetővé más gépek számára az interneten keresztül.
- A webdokumentumokat webkliensek, *böngészők* kérik le a szervertől, és jelenítik meg azokat.
- A kliens és szerver közötti kommunikációt a *HTTP protokoll* biztosítja, ez írja le, hogyan kell egy böngészőnek a kérést, a szervernek pedig a választ elküldenie.

Ezek az elemek szükségesek a web használatához. Ezeken kívül azonban "webesnek" hívunk minden olyan technológiát, amely a fenti elemek bármelyikéhez kapcsolódik. Általában a HTTP fölött zajló kommunikációval vagy a HTML-lel leírt dokumentumokkal kapcsolatos technológiák webesnek számítanak. (Ld. még a **World Wide Web Consortium szabványait** ↪.)

2.2 Kliens-szerver architektúra és a dinamikus webprogramozás

A webes dokumentumok kiszolgálása *kliens-szerver architektúrá*ban történik. A böngésző mint kliens egy HTTP kérést küld a szervernek. A szerver a kérésben foglalt információk alapján összeállítja a HTTP választ, és visszaküldi a böngészőnek. A böngésző a választ feldolgozza, ami általában a válaszban kapott dokumentum megjelenítését jelenti.



A kliens-szerver architektúra

*Statikus weboldalak*ról akkor beszélünk, ha az a tartalom, amit meg szeretnénk jeleníteni már a kérés pillanatában készen áll a szerveren, és a betöltődése után sem változik meg a szerkezete. Ilyenkor a szerver szempontjából is statikus az oldal, hiszen a kikeresett fájlt változatlan formában küldi vissza, és a kliens is statikus, hiszen a megjelenítés után a böngésző nem módosítja az oldal tartalmát.

Dinamikus weboldalakról akkor beszélünk, ha a megjelenített dokumentum előállításához, működtetéséhez, módosításához programot használunk. Mivel az architektúránkban két komponens van, ezért a dinamikusságot mindkét komponens szemszögéből vizsgálhatjuk. *Szerveroldali dinamikus kiszolgálásról* akkor beszélhetünk, ha szerveroldalon a HTML válasz egy program futásának eredményeképpen születik meg. *Kliensoldali dinamizmus* esetén a böngészőben futó program változtatja a megjelenített oldal állapotát.

Ez a tananyag *dinamikus kliensoldali weboldalak programozásáról* szól. Azt mutatja meg, hogy egy betöltött HTML dokumentummal hogyan lehet kapcsolatba lépni a böngészőben futó programmal. Mivel a böngészőkben a *JavaScript* nyelv használható, ezért a HTML oldalak programozását ezen nyelv segítségével végezzük. Mivel kizárólag a böngészőre mint alkalmazáskészítő platformra fogunk koncentrálni, ezért szerverre nem lesz szükség, elég lesz a fájlrendszerből megnyitni az oldalakat. Webszerverre akkor van szükség, ha központilag szeretnénk adatokat elérni, tárolni, vagy szerveroldali dinamikus programokat szeretnénk készíteni.

2.3 A böngészők

A webes világban a böngészők szolgálnak a különböző webes erőforrások megjelenítésére, futtatására. Az erőforrások lehetnek HTML oldalak, képek, stílusállományok, JavaScript programfájlok. A böngésző a HTML oldalakat, képeket megjeleníti, a JavaScript kódot futtatja.

Sokféle böngészőprogram közül lehet választani, ezek közül néhány elterjedtebb:

- **Google Chrome ↪**
- **Mozilla Firefox ↪**
- **Microsoft Edge ↪**

- Opera ↪
- Safari ↪

A böngészőprogramok általában a felhasználói felületükben és az általuk nyújtott szolgáltatásokban térnek el egymástól. Fejlesztés szempontjából az a lényeges, hogy a HTML és CSS állományokat helyesen jelenítsék meg, a JavaScript kódot egységesen futtassák. Szerencsére manapság a böngészők között e tekintetben nincsenek nagy eltérések, ezért bármelyik választható.

2.4 Fejlesztői eszközök a böngészőkben

Mivel a böngésző lesz az alkalmazás-futtató platformunk, meg kell ismerkednünk azokkal az eszközökkel, amelyek a fejlesztést segítik. Az elterjedtebb, népszerű böngészők mindegyike tartalmaz egy ún. fejlesztői eszköztárat, mellyel elérhetjük a programról adott visszajelzéseket és monitorozási adatokat. Ezt az eszközt a legtöbb böngészőprogramban az `F12` billentyű lenyomásával érhetjük el, de a menüben mindig található rá hivatkozás (pl. Google Chrome esetén *További eszközök/Fejlesztői eszközök* menüpont). A fejlesztői eszköztár egyes funkcióit fülek mögé szokták csoportosítani. Nézzük meg a fontosabbakat!

2.4.1 HTML szerkezet vizsgálata

Lehetőség van a betöltött dokumentum szerkezetének vizsgálatára. Itt megjelennek a HTML-ben leírt elemeink. A HTML-fát szabadon böngészhetjük, de általában egy kis nyilacskára kattintva az oldalon is kiválaszthatunk egy elemet, és ilyenkor a fában ez az elem lesz a kijelölt. A fa mellett a kijelölt elem CSS tulajdonságai is megjelennek. Az elemek és a CSS panel is dinamikus, azaz benne bármit megváltoztathatunk, a változások a megjelenített oldalon is megjelennek.

2.4.2 Konzol

A böngészőben futó JavaScript interaktív felülete. Egyrészt itt jelennek meg a kód futása során adódó hibaüzenetek, figyelmeztetések vagy programból kiírt üzenetek, másrészt a konzolba tetszőleges JavaScript kód is beírható, amely az adott fülön megjelenő oldal kontextusában értelmeződik. A konzol remek eszköz próbálgatáshoz, illetve kiválóan használható nyomkövetéshez, hiszen a programban elhelyezett `console.log("üzenet")` paranccsal ide bármikor írathatunk ki információkat.

2.4.3 Forráskód

Az oldalra betöltött JavaScript kódok nézhetőek meg itt, hibakeresésre alkalmas eszköz. A programokba töréspontok helyezhetők el, amelyek futáskor megakasztják a programot. Ekkor az egyes változók értékei lekérdezhetőek (watch), és a program akár lépésenként is végrehajtható.

2.5 Szerkesztők

Az alkalmazásfejlesztéshez megfelelő szerkesztőprogramra is szükség van. A webes dokumentumok forráskódjai egyszerű szövegfájlok. Olyan szerkesztő kell, amelyik képes HTML,

CSS, JavaScript kódot kezelni, és kényelmes, fejlesztőbarát funkciókat nyújt, mint például kódszínezés, kódkiegészítés, automatikus behúzások, projekt kezelése, nyomkövetés, stb. Kétféle lehetőség közül választhatunk: vannak a kisebb méretű, de funkciókban gazdag kódszerkesztők, és vannak az ún. integrált fejlesztőkörnyezetek, amelyek általában nagyobbak, lassabbak, de rengeteg funkcióval rendelkeznek. Mindenki a maga preferenciái szerint választja ki az általa használt eszközt. A webes fejlesztők között az alábbi szerkesztőprogramok a legelterjedtebbek:

- **Microsoft Visual Studio Code ↪** (ingyenes kódszerkesztő)
- **Atom ↪** (ingyenes kódszerkesztő)
- **WebStorm ↪** (oktatási célra ingyenes IDE)

Ezeken kívül az olyan általános szerkesztőprogramok is használhatók, mint pl. a **Notepad++ ↪**.

2.6 Javasolt fejlesztői eszköztár

A könnyebb elindulás érdekében összeállítottunk egy javasolt eszköztárat, mellyel a tananyag végigvihető. A javasolt fejlesztői eszközök multiplatformosak és ingyenesek. Természetesen a fentebb felsorolt eszközök bármelyike alkalmas a tananyag elvégzéséhez.

2.6.1 Javasolt eszközök

- Visual Studio Code szerkesztő
- Google Chrome böngésző

2.6.2 Kezdeti lépések:

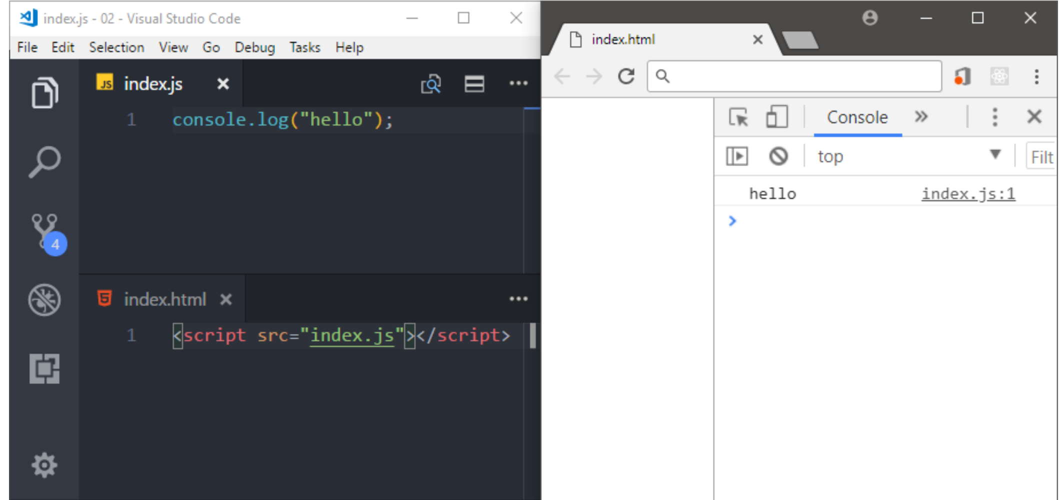
1. Hozzunk létre egy mappát, amelybe dolgozni szeretnénk.
2. Visual Studio Code-ban nyissuk meg a mappát (*File/Open folder…*)
3. Adjunk hozzá egy új fájlt (`index.html`), és valamilyen kezdő tartalommal töltsük fel, mentsük el.
4. Chrome-ban nyissuk meg az `index.html` fájlt. A tartalom megjelenik.
5. Chrome-ban nyissuk meg a fejlesztői eszköztárat (`F12`), válasszuk ki a Konzolt.
6. Visual Studio Code-ban adjunk hozzá egy JavaScript fájlt (`index.js`):

```
console.log("hello");
```

7. Hivatkozzunk erre az `index.html` fájlban:

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

8. Frissítsük az oldalt Chrome-ban (`F5`). A konzolon megjelenik a `hello` szöveg.



A szerkesztő- és a böngészőprogram

3 A JAVASCRIPT PROGRAMOZÁSI NYELV

Nagyfeladat

Célunk, hogy a következő pár fejezet során, apró lépésekben összeépítsünk egy nagyobb méretű játékot, az Aknakeresőt. A feladathoz kapcsolódó kisebb részfeladatokat ehhez hasonló blokkok jelölik majd.

Mint azt az előző fejezetben említettük, a böngészőprogram felelős a HTML oldalon belül a program futtatásáért. A kliensoldali webprogramozásban gyakorlatilag kizárólagosan a *JavaScript programozási nyelv* terjedt el.

3.1 A JavaScript nyelvről általában

A JavaScript egy úgynevezett interpretált szkriptnyelv, ami annyit tesz, hogy a programkód egy futtatókörnyezetben (a mi esetünkben ez a böngésző) fut közvetlenül, fordítás nélkül. A program egy speciális programozási interfészen (API) keresztül kommunikál a böngészőprogrammal, illetve a megjelenített weboldallal. A JavaScript szintaxisában a "C-stílusú" nyelvekhez tartozik, így a vezérlési szerkezetek, nyelvi elemek nagyon hasonlóak a C, C++, Java és C# nyelvek azonos elemeihez.

```
// JavaScript
let x = 1;
for (let i = 2; i < 10; ++i) {
  x = x * i;
}
```

```
// C++
int x = 1;
for (int i = 2; i < 10; ++i) {
  x = x * i;
}
```

JavaScriptben a változókat `let` kulcsszóval hozhatunk létre, a konstansok definiálására a `const` kulcsszót használjuk. Minden utasítást `;` zár, habár ez nem kötelező, de javasolt. Megjegyzéseket a C-stílusú nyelvekből ismert `//` vagy `/* */` módon írhatunk a kódba.

A nyelv további jellemzői:

- **Case-sensitive**, vagyis érzékeny a kis- és nagybetűk közötti különbségekre.
- **Dinamikusan típusos**, vagyis egy változó típusa mindig az éppen aktuális értékének típusától függ, nem lehet a változókat típussal deklarálni.

- **Prototípusos, objektumorientált**, vagyis a nyelvbeli objektumok nem osztályok példányai, hanem más objektumokból (úgynevezett prototípusokból) vannak származtatva.

A nyelvhez részben magyar nyelvű **dokumentáció** ↪ is elérhető.

3.2 Típusok

Mint a legtöbb programozási nyelvben, a JavaScriptben is definiálva vannak bizonyos alapvető típusok. Habár a változóinkat nem típussal hozzuk létre, azok mégis mindig rendelkeznek valamilyen típussal. Ez a típus a változó éppen aktuális értékétől függ. A nyelvben elérhető típusok feloszthatóak *egyszerű* és *összetett* típusokra:

3.2.1 Egyszerű típusok:

- **szám** (`number`): Tetszőleges számérték, lehet egész vagy tizedestört is. Speciális szám értékek a `NaN` (Not A Number), mely érvénytelen matematikai műveletek eredményét reprezentálja (pl. `Math.asin(10)`), illetve az `Infinity` és `-Infinity`, melyek a +/- végtelen értéket jelölik bizonyos végtelen határértékű matematikai műveletek eredményénél (pl. `1/0`). Többféle formátumban megadható, például tizedestörtént (`12.34`), normálalakban (`1.234e2`), nyolcas (`0123`) vagy tizenhatos számrendszerben (`0x123`).
- **szöveg** (`string`): Tetszőleges szöveg érték, melyet idézőjel (`"`), aposztróf (`'`) vagy "backtick" (`` ` ``) szimbólum jelöl. A szövegnek, mint objektumnak léteznek saját tulajdonságai (pl. hossza - `"alma".length`) és műveletei (pl. nagybetűssé alakítás - `"alma".toUpperCase()`). Külön karakter típus nem létezik, a karakterek 1 hosszúságú szövegek a nyelvben.
- **logikai** (`boolean`): Igaz (`true`) és hamis (`false`) értékek. Logikai értékek között az ÉS (`&&`) illetve VAGY (`||`) műveletek definiáltak.
- **nem definiált** (`undefined`): Speciális típus, mely csak egy értéket (`undefined`) vehet fel. Ez az értéke például egy változónak egészen addig, amíg nem adtunk neki értéket.

3.2.2 Összetett típusok:

- **tömb**: Speciális objektum, mely sorszámozottan tartalmaz tetszőleges (akár különböző) típusú elemeket. A tömbök tartalma szabadon változtatható, elemeket tetszőlegesen lehet hozzáadni, törölni, így használható tömb, verem vagy akár sor adatszerkezetként is. Létrehozni az elemek szögletes zárójelben (`[]`) történő felsorolásával lehet:

> **Példa**
>
> ```
> let tomb = [1, true, "alma", []];
> ```
>
> Az elemek elérése szintén `[]` és a 0 kezdőelemű index megadásával lehetséges. `js tomb[2] == "alma"; // true`

- **objektum**: Tetszőleges név-érték párokat tartalmazó adatszerkezet, hasonló pl. a Python vagy a C# nyelv Dictionary típusához. Létrehozni a név-érték párok kapcsos zárójelben történő megadásával lehet:

  ```
  let objektum: { nev: "Anna", eletkor: 18 };
  ```

  Az egyes nevesített mezőket kétféleképpen érhetjük el, vagy a `.` operátorral vagy a `[]` operátorral:

  ```
  objektum.nev == "Anna"; // true
  objektum["eletkor"] == 18; // true
  ```

> **Apróbetűs**
>
> Fontos, hogy a JavaScript az objektumokra és tömbökre mindig cím szerint (nem pedig érték szerint) hivatkozik, így függvényparamétereknél vagy értékadásnál nem jön létre új példány, hanem az eredetivel dolgozunk.
>
> ```
> let objektumA = {
>   ertek: 1
> };
> let objektumB = objektumA;
> objektumB.ertek = 2;
> objektumA.ertek == 2; // true
> ```
>
> Az ebből adódó hibák kikerülésének érdekében érdemes értékadásnál másolni az objektum vagy tömb mezőit, nem pedig közvetlen értékadást használni. Erre használhatjuk az `Object.assign` és az `Array.from` műveleteket.
>
> ```
> // objektumok másolása
> let objektumA = {
>   ertek: 1
> };
> // másolat készítése
> let objektumB = Object.assign({}, objektumA);
> objektumB.ertek = 2;
> objektumA.ertek == 2; // false
>
> // tömbök másolása
> let tombA = [1, 3, 4];
> // másolat készítése
> let tombB = Array.from(tombA);
> tombB[1] = 10;
> tombA[1] == 10; // false
> ```

Figyelem! Az `Object.assign` és az `Array.from` csak az objektum vagy tömb "legfelső szintjén" fog másolást végezni, ha egymásba ágyazott objektumokkal/tömbökkel használjuk, akkor az alsóbb szintek továbbra is cím szerint lesznek hivatkozva! Ebben az esetben kerülőútként a JSON formátumra történő átalakítás, majd visszaalakítás adhat megoldást.

```
let objektumA = {
  ertek: {
    belsoErtek: 1
  }
};
let objektumB = JSON.parse(JSON.stringify(objektumA));
```

3.3 Vezérlési szerkezetek

A JavaScript nyelv vezérlési szerkezetei szinte pontosan megegyeznek a más, hasonlóan C szintaxisú nyelvek azonos elemeivel:

```
for (/* kezdőérték */; /* feltétel */; /* cikluslépés */) {
  // utasítás
}

while (/* feltétel */) {
  // utasítás
}

do {
  // utasítás
} while (/* feltétel */);

if (/* feltétel */) {
  // utasítás
} else {
  // utasítás
}

switch (/* változó */) {
  case /* érték */:
    // utasítás
    break;
  default:
    // utasítás
}
```

Ezeken kívül létezik még a `for` ciklusnak egy változata, mely egy tömb elemein halad végig egyesével.

```
for (let elem of tomb) {
  // utasítás
}
```

3.4 Függvények

A JavaScript nyelvben központi szerepet játszanak a függvények. Ezeket alapvetően kétféleképp lehet definiálni: a `function` kulcsszóval, illetve hozzárendelésként (*arrow function*).

```
// függvény megadása a `function` kucsszóval
function fuggveny(param1, param2) {
  // utasítások
  return /* visszatérési érték */;
}

// Például
function osszead(x, y) {
  return x + y;
}

// Hívása
osszead(10, 32); // --> 42
```

Mint a fenti példa mutatja, a függvények paramétereit nem kell típusokkal ellátni, azok automatikusan az átadott paraméterek típusát veszik fel. Szintén nem kell megadni a függvény visszatérési típusát, csupán a `return` kulcsszóval megadni a visszatérési értéket.

Egy másik lehetőség függvény megadására az úgynevezett *arrow function*, vagyis a "hozzárendeléses" megadás. Ez a megadás akkor lehet hasznos, amikor egy másik függvénynek kell megadni egy olyan paramétert, mely maga is függvény. Az *arrow function*-ök használata akkor eredményez szép, tiszta kódot, ha a bemeneti paraméterek alapján a kimeneti érték egy zárt kifejezés formájában megadható.

```
// függvény megadása hozzárendelésként
let osszead = (x, y) => (x + y);
```

A hozzárendeléses megadást leginkább a tömbök saját függvényeinél tudjuk hatékonyan használni. Ezek a saját függvények (`map`, `filter`, `some`, `every`, stb.) teljes programozási tételek megvalósítását teszik lehetővé egy rövid kifejezés formájában.

A `map` segítségével leképezhetjük valamilyen hozzárendelés szerint egy tömb elemeit, míg a `filter`-rel kiszűrhetünk valamilyen tulajdonságú elemeket. A szűrés paramétere egy olyan hozzárendelés, ami a megtartandó elemekhez `true`, az eldobandó elemekhez `false` értéket rendel. Hasonló hozzárendelést kell paraméterül adni a `some` és az `every` függvényeknek, melyek az eldöntés programozási tétel két formáját valósítják meg. A `some` esetében azt vizsgáljuk, hogy van-e olyan elem, amire a hozzárendelésünk `true` értéket ad, míg az `every` esetében azt, hogy minden elem ilyen-e.

Példa

```
const tomb = [1, 2, 4, 1, 6, 2, 5, 3];
// minden elem megszorzása 2-vel
tomb.map(x => x * 2);         // [2, 4, 8, 2, 12, 4, 10, 6]
// páros elemek kiválogatása
tomb.filter(x => x % 2 == 0); // [2, 4, 6, 2]
// van-e 10-nél nagyobb elem
tomb.some(x => x > 10);       // false
// minden elem 10-nél kisebb-e
tomb.every(x => x < 10);      // true
```

Apróbetűs

Típusos alternatívák

A JavaScript nyelv az utóbbi években egyre nagyobb népszerűségre tett szert. Ennek hatásaként számos olyan programozási nyelv jelent meg, mely igyekszik kibővíteni a JavaScript által nyújtott lehetőségeket. Egyik lehetséges iránya ennek a bővítésnek a statikus típusellenőrzés bevezetése, vagyis hogy a változók, függvényparaméterek és a függvények visszatérési értékének előre meghatározott típust adunk. A statikus típusellenőrzés számos előnnyel rendelkezik, pedagógiai szempontból például erősíti a típusfogalom megértését, valamit a programbéli függvények matematikai függvényekkel való párhuzamát.

Több nyelvváltozat is létezik, melyek statikus típusellenőrzéssel egészítik ki a JavaScript nyelvet. A legismertebbek a **TypeScript** ↪ és a **Flow** ↪.

3.5 A JavaScript kipróbálása

A programozási nyelv kipróbálásához számos remek eszköz áll rendelkezésünkre. Legegyszerűbb ilyen eszköz maga a böngészőben található *konzol*, melyet a fejlesztői eszközök (`F12`) között találhatunk meg. A konzolba írt minden utasítást a JavaScript értelmező azonnal futtatni képes.

> **Apróbetűs**
>
> A fejlesztői eszközök, mint a konzol bármilyen oldalon megnyithatók, de a kísérletezéshez érdemes egy üres böngészőablakot használni. Ilyen üres böngészőablakot Google Chrome esetében a címsorba írt `about:blank` szöveggel nyithatunk.

Az egyik legegyszerűbb utasítás maga a konzolra történő kiírás, a `console.log`.

```
console.log("Hello világ");
```

Szintén lehetőségünk van változók létrehozására a `let` kulcsszóval. Egyszerűbb, böngészőben történő beolvasásra használhatjuk a `prompt` függvényt, ami egy felugró ablakban vár egyszerű szöveges bemenetet. A `prompt` párja az `alert`, mellyel egy rövid szöveget jeleníthetünk meg felugró ablakban.

```
let nev = prompt("Add meg a neved");
alert("Szia, " + nev);
```

Az `alert` és a `prompt` leginkább kísérletezésre, hibakeresésre alkalmas függvények, ezért valós programban kerüljük ezek használatát. A `console.log` művelet szintén alkalmas hibakeresés céljából történő kiírásokra, hiszen a böngészőablakban nem, csak a konzolon látszik a kimenete.

Mivel a konzolba egyszerre csak egy utasítást írhatunk be, ezért ha hosszabb kódokkal szeretnénk kísérletezni, akkor már érdemesebb valamilyen szerkesztőprogramot segítségül hívni. Számos olyan online eszköz létezik, melyek segítségével azonnal futtathatjuk az általunk írt JavaScript kódot, melynek az eredménye is azonnal megjelenik. Néhány ilyen eszköz:

- **Repl.it ↪**
- **JSBin ↪**

3.6 Feladatok

1. Készíts programot, ami beolvas egy számot és eldönti, hogy az páros vagy páratlan! Használd a maradék (`%`) operátort!

   ```
   const szam = parseInt(prompt("Adj meg egy egész számot!"));
   if (szam % 2 === 0) {
     alert("Páros");
   } else {
     alert("Páratlan");
   }
   ```

2. Egy tömbben adott számoknak a sorozata, adjuk meg az összegüket!

```
const tomb = [1, 4, 12, 4, -5];
let osszeg = 0;
for (let szam of tomb) {
  osszeg += szam;
}
console.log(osszeg);
```

3. Készíts függvényt `faktorialis` néven, ami egy ciklussal kiszámítja az `n` faktoriális értékét!

```
function faktorialis(n) {
  let eredmeny = 1;
  for (let i = 2; i <= n; i++) {
    eredmeny *= i;
  }
  return eredemeny;
}
```

4. Készíts függvényt, ami egy tömb elemei közül megszámolja, hogy hány darab `x` érték található!

```
function darab(tomb, x) {
  let darab = 0;
  for (let elem of tomb) {
    if (elem === x) {
      darab++;
    }
  }
  return darab;
}
// vagy
function darab(tomb, x) {
  return tomb.filter(elem => elem === x).length;
}
```

5. **Nagyfeladat**

   Készíts függvényt, ami pontosan 3 számot kap paraméterül, és megadja, hogy az első paraméterre igaz-e, hogy a második és a harmadik között található (határokat is beleértve).

6. **Nagyfeladat**

   Készíts függvényt, ami adott minimum és maximum érték között (határokat beleérve) állít elő egy véletlenszerű egész számot! Megoldásodhoz használd a `Math.random` ↪ függvényt!

3.7 Adatszerkezetek ábrázolása

Az előzőekben megnéztük, hogyan lehetséges sorozat jellegű adatszerkezeteket (tömb, objektum) létrehozni JavaScriptben. Valós alkalmazásokban ezeknél bonyolultabb adatszerkezetekre is szükségünk lehet, de JavaScriptben minden ilyen bonyolultabb adatszerkezetet le tudunk írni tömbök és objektumok segítségével, ezek egymásba ágyazásával.

Hogyan is ágyazhatók egymásba tömbök és objektumok?

- A tömb egy vagy több eleme maga is egy tömb
- A tömb egy vagy több eleme egy objektum
- Egy objektum egy vagy több mezője egy tömb
- Egy objektum egy vagy több mezője maga is egy objektum

Ezen módszerekkel akár nagyon nagy bonyolultságú adatok is leírhatóak. Nézzünk egy példát!

Példa

Készítsünk egy adatszerkezetet, ami egy iskolának és annak osztályainak adatait tartalmazza!

Kiindulásképp vegyünk egy objektumot, melynek a mezői az iskola alapvető adatait tartalmazzák:

```
{
  nev: "JavaScript Általános Iskola",
  cim: "1337 Világháló utca 404.",
  om: "528272478"
}
```

Ez az adatszerkezet tovább bővíthető, ha egy mezőn belül egy tömbben eltároljuk, hogy milyen osztályok vannak az iskolában:

```
{
  nev: "JavaScript Általános Iskola",
  cim: "1337 Világháló utca 404.",
  om: "528272478",
  osztalyok: [
    "1.a",
    "1.b",
```

```
      "2.a",
      "2.b"
    ]
  }
```

Ebben a példában csak az osztályok nevét tároljuk el, de semmi egyebet nem tudunk az osztályról. Ha további információkat szeretnénk tárolni (pl. osztálylétszám, osztályfőnök), akkor megtehetjük, hogy minden egyes osztályt egy objektum reprezentál az adatszerkezetünkben:

```
{
  nev: "JavaScript Általános Iskola",
  cim: "1337 Világháló utca 404.",
  om: "528272478",
  osztalyok: [
    { nev: "1.a", ofo: "Dan Abramov",  letszam: 14 },
    { nev: "1.b", ofo: "Eric Elliot",  letszam: 16 },
    { nev: "2.a", ofo: "David Walsh",  letszam: 12 },
    { nev: "2.n", ofo: "Kyle Simpson", letszam: 13 }
  ]
}
```

Ez a példa már kellőképpen összetett adatszerkezetet mutat be, de természetesen még ez is tovább bővíthető lenne, például ha minden osztályhoz egy tömbben felsorolnánk minden diákot, akiket külön-külön egy-egy objektum reprezentálhatna.

Apróbetűs

A példákban látott formátum ihlette az úgynevezett **JSON** ↪ adatleíró formátumot, melyet leginkább programok interneten keresztüli kommunikációjához használnak. A JSON formátum alapvetően megegyezik a JavaScript nyelv tömb-objektum leíró formátumával, habár szigorúbb megszorítások érvényesek rá (pl. minden mezőnevet idézőjelek közé kell tenni, az aposztróf nem használható a szövegek jelölésére és a tömb utolsó eleme után nem lehet vessző), minden érvényes JSON adatszerkezet egy az egyben érvényes JavaScript adatszerkezet is egyben.

3.8 Feladatok

1. Készíts olyan adatszerkezetet, melybe egy bevásárlólista információit tárolhatjuk. A listán többféle dolgot szeretnénk tárolni, és mindegyik elemről tudni szeretnénk, hogy mi az és mennyit szeretnénk belőle vásárolni.

```
const bevasarloLista = [
  { mit: "alma", mennyit: 1, mertekEgyseg: "kg"},
  { mit: "liszt", mennyit: 2, mertekEgyseg: "kg"},
  { mit: "tej", mennyit: 6, mertekEgyseg: "l"},
  { mit: "sonka", mennyit: 25, mertekEgyseg: "dkg"}
];
```

2. Készíts egy olyan N×M-es mátrix adatszerkezetet (tömbök tömbje), melynek cellái 1, 2, 3 számokat tartalmaznak. Készítsünk programot, ami megszámolja, hogy melyik számból hány darab van a mátrixban!

```
const matrix = [
  [1, 3, 3],
  [1, 1, 2],
  [3, 2, 2]
];

const darabok = {
  "1": 0,
  "2": 0,
  "3": 0
};

for (let sor of matrix) {
  for (let ertek of matrix) {
    darabok[ertek] += 1;
  }
};
```

3. **Nagyfeladat**

   Készíts függvényt, ami paraméterül kap egy N és egy M számot, valamint egy tetszőleges kezdőértéket és eredményül ad egy olyan N×M-es mátrixot, aminek minden cellájában a paraméterül kapott kezdőérték van.

4. **Nagyfeladat**

   Készíts olyan függvényt, amely paraméterül kap egy mátrixot és egy x, y koordinátapárt, és a megadott cella (x, y koordináták alapján) összes szomszédjának növeli eggyel az értékét.

Letöltés

⭳ **A fejezethez tartozó nagyfeladatok megoldásai ↪**

Apróbetűs

Objektumorientált programozás JavaScriptben

A JavaScript nyelv újabb verziói (EcmaScript 2015) már támogatják a klasszikus objektumorientált minták használatát is. Ennek megfelelően létrehozhatók osztályok a `class` kulcsszóval, melyek egymásból származtathatók is. Ez olyan tanulók esetében lehet érdekes, akik korábban már dolgoztak valamilyen objektumorientált nyelvvel, például Java-val vagy C#-pal. Az osztályok használatáról részletesebben a nyelv **dokumentációjában ↪** olvashatunk.

4 FELÜLETI ELEMEK PROGRAMOZÁSA

Interaktív alkalmazások fejlesztése során mindig van egy *felhasználói felület*, amin keresztül a felhasználó képes az alkalmazással kapcsolatba lépni: információkat megtekinteni és adatokat megadni. A böngésző esetén a felhasználói felület maga a megjelenő weboldal, amit HTML és CSS nyelv segítségével írunk le. A HTML nyelvvel megadjuk, hogy milyen elemekre milyen szerkezetben van szükségünk, a CSS nyelvvel pedig ezek megjelenését határozzuk meg. A felhasználói felület működtetésére, az adatok feldolgozására azonban szükség van egy *program*ra is, amit a böngészőkben JavaScript nyelven tudunk megírni. A HTML elemek nem programozhatóak közvetlenül, hanem egy *programozási interfész*en keresztül érjük el őket, amit *Dokumentum Objektum Modell*nek, röviden DOM-nak nevezünk.

4.1 A JavaScript kód helye a HTML dokumentumban

JavaScript kódot a `<script>` elem segítségével lehet az oldalon elhelyezni. Egy oldalon belül akárhány `<script>` elem használható az oldal bármely pontján. Tipikusan azonban két helyen jelenik meg:

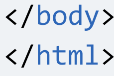
- általában közvetlenül a `</body>` előtt, mivel itt már minden felületi elem betöltődött és elérhető;
- a `<head>` részben, ahol főleg olyan kódokat helyeznek el, amelyek az adatok előkészítését végzik, és nincs szükségük az oldal felületi elemeinek elérésére.

Helyzetét tekintve egy JavaScript kód lehet:

- *belső*, a HTML forráskódban megjelenő; vagy
- *külső* állományként betöltött; ez a jellemző.

Példa

```
<!DOCTYPE html>
<html>
<head>
  <title>A JavaScript kód helye</title>
  <!-- belső szkript -->
  <script>
    const MAXN = 100;
  </script>
</head>
<body>
  <h1>Hello világ!</h1>
  <!-- külső szkript -->
  <script src="kulso.js"></script>
```

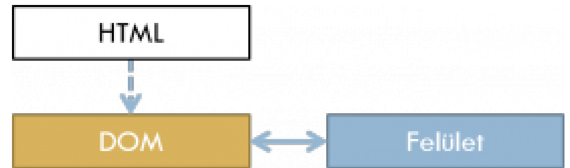


```
</body>
</html>
```

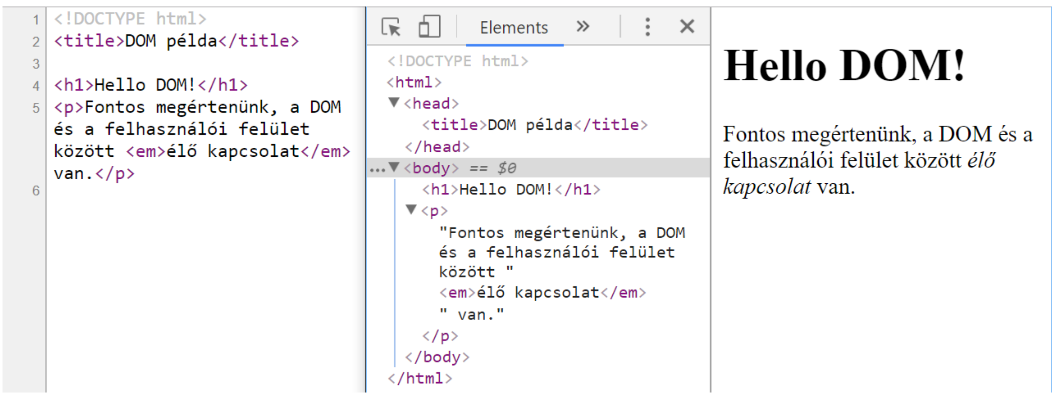
4.2 A Dokumentum Objektum Modell (DOM)

Egy HTML nyelven írt állomány nem más, mint egy szöveges dokumentum. A böngészőnek ahhoz, hogy ezt a szöveges információt megjelenítse, egy belső ábrázolást kell készítenie a szöveges HTML elemekből. Ez a *belső ábrázolás* a Dokumentum Objektum Modell, röviden DOM.

Az oldal betöltése során a böngésző megkapja a szöveges HTML állományt, és elkezdi a benne lévő HTML elemeket feldolgozni. Minden egyes HTML elemhez létrehoz egy JavaScript objektumot, és ezeket a JavaScript objektumokat ugyanolyan fa hierarchiába szervezi, ahogy azok az eredeti HTML dokumentumban is szerepelnek. Az így kialakult *JavaScript objektumhierarchia* a DOM. Utolsó lépésként a böngésző a DOM alapján megjeleníti a böngészőben a HTML elemeknek megfelelő weboldalt.



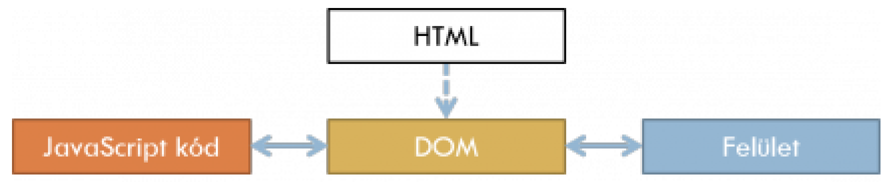
A HTML kód, a DOM és a felhasználó felület összefüggése



Az oldal forráskódja (balra), a belőle felépült DOM-fa (középen) és a megjelenített felület (jobbra)

Fontos megértenünk, hogy a DOM és a felhasználói felület között *élő kapcsolat* van. Az oldal megjelenítése mindig a DOM alapján történik, illetve a felületi változások mindig tükröződnek a DOM-ban is. A szöveges HTML állományra csupán egyszer, a folyamat elején van szükség, a böngésző minden további műveletet a HTML-ből felépített DOM-on végez el.

A DOM azonban nemcsak egy olyan belső ábrázolása a HTML elemeknek, ami alapján a felület kirajzolása történik, hanem JavaScript objektumai révén *programozási felületet* is nyújt az oldal kontextusában futó JavaScript kódoknak. Ez annak köszönhető, hogy a DOM JavaScript objektumai elérhetőek a JavaScript kódból, és szabadon manipulálhatók: tulajdonságaik lekérdezhetőek és beállíthatóak, metódusai meghívhatóak.



A JavaScript kód interakciója a DOM-mal

A DOM tehát az a programozási interfész, amin keresztül a JavaScript kód a felhasználó felülethez hozzáfér. Segítségével tudunk adatokat kinyerni a felületből vagy információkat megjeleníteni a felületen. A JavaScript kód számára tehát a DOM jelenti a *bemeneti-kimeneti interfészt*.

> **Apróbetűs**
>
> Ha a böngészőben megtekintjük az oldal forrását, akkor a betöltött, szöveges HTML állományt látjuk. A fejlesztői eszköztár "HTML elemek" fülén viszont a DOM struktúrát böngészhetjük. A kettő jelentősen el is térhet egymástól, ha JavaScript kód a DOM fát megváltoztatja.

4.3 DOM műveletek

A DOM programozása tipikusan két lépésen keresztül történik:

1. Megfelelő elem(ek) kiválasztása
2. A kiválasztott elem(ek) használata (olvasás/írás)

4.3.1 Elemek kiválasztása

Elemeket legegyszerűbben a következőképpen lehet kiválasztani:

- *azonosítójuk alapján* (`id` attribútum): `document.getElementById(azon)`
- *CSS szelektorral*:
  - egy elem: `document.querySelector(szel)`
  - több elem: `document.querySelectorAll(szel)`

```
<form>
  Név:
  <input id="nev" value="Frodó">
  <button>Nyomj meg!</button>
</form>
<script>
  console.log( document.getElementById("nev") );
  console.log( document.querySelector("#nev") );
  console.log( document.querySelectorAll("form > *") );
</script>
```

Ismétlés

HTML elemek kiválasztása CSS szelektorokkal

CSS-ben **HTML elemek kiválasztása** ↪ ún. *szelektorok*kal lehetséges:

- az elem nevével (pl. `button`),
- az azonosítójával (pl. `#fejlec`),
- az alkalmazott stílusosztály alapján (pl. `.fontos`),
- az attribútumaival (pl. `[name=nev]`),
- mindenkit kiválasztva (`*`), illetve
- ezek kombinálásával (pl. `input.hibas[type=text]`)

Lehetőség van továbbá hierarchikus viszonyokat is megadni:

- szülő-gyerek (pl. `form > input`)
- szülő-leszármazott (pl. `#torzs div`)
- közvetlenül utána jövő testvér (pl. `p + span`)
- utána jövő testvérek (pl. `p ~ span`)

Apróbetűs

Másik lehetőség elemek kiválasztására a DOM-fa bejárása. A fa gyökere a `document` objektum. Általában azonban a fenti módszerrel kijelölünk egy elemet (`elem`), majd onnan három irányba tudunk ellépni:

- szülő felé: `elem.parentNode`
- testvérek felé: `elem.nextElementSibling`, `elem.previousElementSibling`
- gyerekek felé: `elem.children`, `elem.firstElementChild`, `elem.lastElementChild`, `elem.childElementCount`

A `$` segédfüggvény

Viszonylag gyakran kell elemeket kiválasztanunk a felület programozása során. A fenti hosszú kiválasztó műveleteket becsomagolhatjuk rövidebb formába is, így gyorsabban és hibamentesebben tudunk elemeket kijelölni.

```
function $(szelektor) {
  return document.querySelector(szelektor);
}

const elem = document.querySelector("#azonosito");
// helyett
const elem = $("#azonosito");
```

A tananyag további részében feltételezzük, hogy a `$` függvény adott az oldal kontextusán belül.

4.3.2 Egy kiválasztott elem tulajdonságai

Egy kiválasztott DOM objektum használatához ismerni kell azt, hogy annak milyen tulajdonságai és metódusai vannak. A *tulajdonságok*at illetően szerencsére egyszerű helyzetben vagyunk: többségük egy egyszerű átírási szabályt követve a megfelelő HTML elem attribútumai nevének felel meg. Az *átírási szabály* (camel-case): minden tulajdonság kisbetűvel kezdődik, szóösszetétel határán a következő szó nagybetűvel írandó.

Példa

Vegyük példának az `input` elemet:

HTML ATTRIBÚTUM	DOM TULAJDONSÁG
`type`	`type`
`value`	`value`
`readonly`	`readOnly`
`maxlength`	`maxLength`

Mindegyik DOM elem fontos tulajdonsága az `innerHTML` adattag, amelyen keresztül lekérdezhető vagy beállítható az adott elem nyitó- és záróeleme közötti rész-HTML.

Apróbetűs

Egy DOM objektumnak további tulajdonságai és metódusai vannak. Ezekről részletesen a **dokumentációban** ↪ lehet olvasni (pl. **az `input` elem leírása** ↪).

4.4 A DOM mint bemenet/kimenet

Ahogy fentebb már írtuk, a felhasználó a felhasználói felületen keresztül lép kapcsolatba a programmal: azon jelennek meg információk vagy adhatók meg adatok. A felhasználói felület programbeli elérése a DOM-on keresztül lehetséges. A JavaScript program számára tehát a DOM szolgál bemenetként és kimenetként.

Egy tipikus JavaScript program általános felépítése hasonlít a konzolos alkalmazásokéhoz:

- beolvasás (a DOM-ból)
- feldolgozás (bemenettől és kimenettől függetlenül)
- kiírás (kiírás a DOM-ba)

A feldolgozás – mivel bemenettől és kimenettől független – pusztán a JavaScript nyelvi elemeivel (adatszerkezetekkel és vezérlési szerkezetekkel) megoldható.

4.4.3 Beolvasás a DOM-ból

Beolvasni általánosságban annyit jelent, hogy a megfelelő elem DOM objektumának megfelelő tulajdonságát lekérdezzük.

1. **Példa**

   Szöveges beviteli mezőbe írt érték beolvasása: A szöveges beviteli mező értékét a `value` attribútummal tudjuk HTML-ben beállítani. Az ennek megfelelő `value` tulajdonság szolgál a lekérdezésére is.

```
<input id="nev">
<script>
  const nevDomElem = document.querySelector("#nev");
  const nev = nevDomElem.value;
  console.log(nev);

  // rövidebben, a $() segédfüggvény használatával

  console.log( $("#nev").value );
</script>
```

2. **Példa**

   Jelölőmező értékének kiolvasása: HTML-ben a jelölőmező értékét a `checked` attribútummal lehet beállítani, a DOM-ban ugyanilyen nevű tulajdonság szolgál lekérdezésére is.

   ```
   <input type="checkbox" id="elfogad" checked>
   <script>
     const elfogad = $("#elfogad").checked;
   </script>
   ```

3. **Példa**

   Egy link hivatkozásának kiolvasása: A link hivatkozását a `href` attribútumon keresztül tudjuk beállítani. A DOM-ban ugyanilyen néven szerepel a neki megfelelő tulajdonság is.

   ```
   <a href="http://webprogramozas.inf.elte.hu">Webprogramozás az
   ELTÉn</a>
   <script>
     const href = $("a").href;
   </script>
   ```

4.4.4 Kiírás a DOM-ba

A DOM objektumok tulajdonságai nemcsak lekérdezhetők, hanem beállíthatók is. A DOM objektumban történt változások azonnal tükröződnek a felületen is.

1. **Példa**

   Kép forrásának beállítása: Az `img` elemnek az `src` attribútuma szolgál forrásának megadására. Az ugyanilyen nevű tulajdonság beállításával programozottan végezhetjük el a beállítást.

   ```
   <img src="" id="kep">
   <script>
     const url = "http://kepek.hu/alma.png";
     $("#kep").src = url;
   </script>
   ```

2. **Példa**

   Választóelem bejelölése: Akárcsak a jelölőmezőnél, a rádiógomboknál is a `checked` attribútum vezérli a kijelölés állapotát. A megfelelő DOM objektumnál az ugyanilyen nevű tulajdonságnak kell igaz értéket adnunk.

   ```
   <input type="radio" name="nem" value="ferfi" checked> férfi
   <input type="radio" name="nem" value="no"> nő
   <script>
     $("[name=nem][value=no]").checked = true;
   </script>
   ```

4.4.5 Új elemek létrehozása

A kiírás egy speciális formája, amikor új HTML elemeket szeretnénk az oldalon megjeleníteni. Ezt legegyszerűbben úgy tehetjük meg, ha egy elem nyitó és záróeleme közötti részébe szöveges HTML formában adjuk meg az új tartalmat. Ezt az elem `innerHTML` tulajdonságának beállításával tehetjük meg. Ezzel a módszerrel tetszőleges mennyiségű elem létrehozható.

Példa

Írjunk ki üdvözlő szöveget címsorként az oldalra! Ehhez a `kimenet` azonosítójú `<div>` elem `innerHTML` tulajdonságának adjuk értékül a megjelenítendő tartalmat.

```
<div id="kimenet"></div>
<script>
  const udvozles = "<h1>Hello mindenki</h1>";
  $("#kimenet").innerHTML = udvozles;
</script>
```

Apróbetűs

HTML elemeket elemibb DOM műveletekkel is hozzáadhatunk az oldalhoz, finoman hangolva az új elemek létrehozásának folyamatát.

- `document.createElement(HTMLElemNév)` : létrehoz a paraméterben megadott `HTMLElemNév` -nek megfelelő DOM objektumot a memóriában.
- `szülőElem.appendChild(gyerekElem)` : a `szülőElem` gyerekeihez utolsóként hozzáadja a `gyerekElem` DOM objektumot.

Példa

Adjunk egy új listaelemet a felsoroláshoz!

```
<ul id="lista">
  <li>első</li>
  <li>második</li>
</ul>
<script>
  const ujListaElem = document.createElement("li");
  ujListaElem.innerHTML = "harmadik";
  $("#lista").appendChild(ujListaElem);
</script>
```

Elemek elhelyezésére, mozgatására, beszúrására **további DOM műveletek** ↪ állnak rendelkezésre (`insertBefore` , `removeChild` , `replaceChild`).

4.5 HTML elemek szöveges generálása

Kiírás során gyakran kell sok elemet létrehoznunk. Ennek egyik lépése annak a HTML szövegnek a létrehozása, amelyet aztán a célelem `innerHTML` -jének értékül adunk. A HTML szöveg generálásához a JavaScript sablonszöveg operátora (`` ` ` ``) ad elegáns megoldást.

4.5.1 Rövid statikus szöveg megadása

Példa

```
const s = `<h1>Hello Gandalf!</h1>`;
```

4.5.2 Többsoros statikus szöveg megadása

Példa

```
const s = `
  <div>
    <p>I am your <strong>father</strong>, Luke!</p>
  </div>
`;
```

4.5.3 Változók behelyettesítése

Példa

```
const pontszam = 100;
const s = `Összesen ${pontszam} pontot gyűjtöttél!`;
```

4.5.4 Tömbök kiírása leképezéssel

Példa

```
const nevek = ["Sára", "Zsófi", "Dávid", "Matyi", "Veronika"];
const s = `
  <ul>
    ${nevek.map(nev => `
      <li>${nev}</li>
    `).join("")}
  </ul>
`;
```

4.5.5 Feltételes kiírás (elágazás)

Példa

```
const homerseklet = 5;
const s = `
  <span>Hú de nagyon
  ${homerseklet > 20 ? "meleg" : "hideg"}
  van</span>
`;
```

4.5.6 Függvényekkel

Példa

```
function lista(szovegTomb) {
  return `
    <ul>
      ${szovegTomb.map(e =>
        listaElem(e)
      ).join("")}
    </ul>
  `;
}
function listaElem(s) {
  return `<li>${s}</li>`;
}

const nevek = ["Sára", "Zsófi", "Dávid", "Matyi", "Veronika"];
const s = lista(nevek);
```

Vagy rövidebben a `lista` függvényt:

```
function lista(nevek) {
  return `
    <ul>
      ${nevek.map(listaElem).join("")}
    </ul>
  `;
}
```

4.6 Stílusok módosítása

Ismétlés

HTML-ben egy elem megjelenését a `class` vagy a `style` attribútumon tudjuk vezérelni CSS-sel.

```
<div class="fontos kiemelt" style="position: absolute; top:
50px;">Aragorn</div>
```

4.6.1 Stílusattribútum programozása

A `style` attribútumot a DOM objektum `style` tulajdonságán keresztül érhetjük el. Ez egy olyan objektumot ad vissza, amelynek tulajdonságai az egyes CSS stílustulajdonságoknak felelnek meg a camel-case átírás szabályait követve (pl. `border-top-left-radius: 20px` kódbeli megfelelője a `style.borderTopLeftRadius = "20px"`). Lekérdezésre ritkábban használjuk, általában gyakran változó stílustulajdonságokat (pl. pozíció) állítunk be vele.

Példa

Átírási szabályok

CSS STÍLUSTULAJDONSÁG	`STYLE` OBJEKTUM TULAJDONSÁGA
`left`	`left`
`background-color`	`backgroundColor`
`border-bottom-width`	`borderBottomWidth`
`border-top-left-radius`	`borderTopLeftRadius`

Példa

Állítsuk be egy `<div>` elemnek a pozíció tulajdonságát HTML attribútumon keresztül, a `top` , `left` tulajdonságát JavaScriptből:

```
<div style="position: absolute" id="mozgo_elem"></div>
<script>
  $("#mozgo_elem").style.top  = "25px";
  $("#mozgo_elem").style.left = "42px";
</script>
```

Apróbetűs

A `style` objektumon keresztül tetszőleges stílustulajdonság beállítható, de csak azok kérdezhetők le, amelyek a `style` attribútumon keresztül vagy JavaScriptből lettek beállítva. A többi tulajdonság egyszerűen üres szöveges értéket tartalmaz. (Ld. pl. a fenti példában: `console.log($("#mozgo_elem").style)`)

Ha kíváncsiak vagyunk egy HTML elem tetszőleges stílustulajdonságának aktuális értékére, akkor azt az ún. **számított stíluson** ↪ keresztül lehet lekérdezni a `window.getComputedStyle(DOM_objektum)` metódus segítségével:

```
const elem = $("#mozgo_elem");
const szamitott_stilus = window.getComputedStyle(elem);
console.log(szamitott_stilus);
console.log(szamitott_stilus.borderBottomWidth);
```

4.6.2 Stílusosztály programozása

A HTML elem `class` attribútumát az elemnek megfelelő DOM objektum `classList` tulajdonságán keresztül tudjuk programozni. Ez a beállított stílusosztályok gyűjteményét adja vissza, és többek között a következő hasznos metódusokat szolgáltatja:

- `add(osztály)` : hozzáadja az `osztály` szöveget a stílusosztályokhoz;
- `remove(osztály)` : eltávolítja az `osztály` stílusosztályt a többi közül;
- `toggle(osztály)` : ki-bekapcsolja az `osztály` stílusosztályt jelenlététől függően (ha nincs, akkor hozzáadja, ha van, akkor kiveszi).

Példa

Tegyük `fontos` -sá a harmadik listaelemet!

```
<style>
  .fontos {
    color: red;
    border: 2px solid orange;
  }
</style>
<ul>
  <li>első</li>
  <li>második</li>
  <li>harmadik</li>
  <li>negyedik</li>
</ul>
<script>
  $("ul > li:nth-child(3)").classList.add("fontos");
</script>
```

Apróbetűs

Egy HTML elem `class` attribútumának értékét a `className` tulajdonsággal is elérhetjük, ami szöveges formában adja vissza a stílusosztályokat. Több stílusosztály esetén szóközzel elválasztva adja vissza, illetve várja az értéket.

```
console.log(elem.className);
elem.className = "fontos kiemelt";
```

4.7 Feladatok

1. Másold át az értesítési címet a számlázási címbe!

```
Értesítési cím:
<input id="ertesitesi_cim" value="1111 Budapest, Nekeresd utca
11.">
Számlázási cím:
<input id="szamlazasi_cim">

<script>
  // beolvasás
  const ertesitesi_cim = $("#ertesitesi_cim").value;
  // kiírás
  $("#szamlazasi_cim").value = ertesitesi_cim;
</script>
```

2. Csak akkor kérd be a leánykori nevet, ha nő az illető!

```
<input type="radio" name="nem" value="ferfi" checked> férfi
<input type="radio" name="nem" value="no"> nő
Leánykori név: <input id="leanykori_nev">
<script>
  // beolvasás
  const no = $("[name=nem][value=no]").checked;
  // kiírás
  $("#leanykori_nev").hidden = !no;
</script>
```

3. Listázd ki az oldal összes hiperhivatkozásának a címét!

```
<a href="http://www.elte.hu">ELTE</a>
<a   href="http://webprogramozas.inf.elte.hu">Webprogramozás   az
ELTÉn</a>
<a href="http://www.inf.elte.hu">ELTE Informatikai Kara</a>
<ul id="hivatkozasok"></ul>

<script>
  function lista(szovegTomb) {
    return szovegTomb.map(e =>
      `<li>${e}</li>`
    ).join("")
  }

  // beolvasás
  const linkek = Array.from( document.querySelectorAll("a") );
  const hivatkozasok = linkek.map(a => a.href);
  // kiírás
  $("#hivatkozasok").innerHTML = lista(hivatkozasok);
</script>
```

4. **Nagyfeladat**

   Ismert N értéke egy beviteli mezőben. Készíts egy NxN-es táblázatot!

Letöltés

- ⭳ **A fejezethez tartozó nagyfeladatok megoldásai ↪**

5 INTERAKTÍV PROGRAMOK – ESEMÉNYKEZELÉS

Az előző fejezetben láttuk, hogy egy JavaScript program hogyan tud kapcsolatba lépni a felületi elemekkel. Az ott látott programok az oldal betöltődésekor futottak le, a felhasználónak további beleszólása nem volt az alkalmazásba. Ebben a fejezetben azt nézzük meg, hogy hogyan tud a felhasználó kapcsolatba lépni az alkalmazással.

Az alkalmazásoknak általában fontos része a *felhasználói interakció*. Ilyen az például, amikor a felhasználó a felületen megad adatokat, egy gomb lenyomásával feldolgozást kezdeményez vagy egérrel irányít egy játékot. Általánosan tekintve: a felhasználó tevékenységére az alkalmazás valahogyan reagál. Ez a működési mód alapjaiban tér el a konzolos alkalmazásokétól, amelyek általában lineárisan, előre meghatározott sorrendben futnak le. A böngészőbe betöltött oldal esetében azonban a felhasználói tevékenységek hatására kis részprogramok hajtódnak végre. A felhasználói tevékenység ún. *eseményeket* vált ki az oldalon, az erre válaszul lefutó részprogramokat pedig *eseménykezelőknek* hívjuk. Az alkalmazás egésze igazából nem más, mint ezeknek az eseménykezelőknek a laza halmaza. Ezt a programozási modellt *eseményvezérelt programozásnak* is nevezik.

A felhasználó tehát eseményeken és eseménykezelőkön keresztül lép kapcsolatba az alkalmazással. Ebben a fejezetben azt nézzük meg, hogy milyen eseményeket tud a felhasználó kiváltani, és hogyan lehet JavaScriptben eseménykezelőket írni.

5.1 Az eseménykezelő függvények mint programok

Egy eseményvezérelt programban nincsen a konzolos alkalmazásoknál megismert egy nagy belépési pont, ahol beolvasunk, feldolgozunk és kiírunk, majd a program véget ér. Ehelyett több belépési pont van az alkalmazásban, amelyeket a felhasználói események, pontosabban az ezekre reagáló eseménykezelő részprogramok képviselnek. Minden egyes eseménykezelő egy *kis program* önmagában, amely a szokásos lépéseket hajtja végre:

- *beolvasás*: első lépésként beolvassa az adatokat;
- *feldolgozás*: a bemeneti adatokból előállítja a kimeneti adatokat;
- *kiírás*: a kimeneti adatokat megjeleníti a felhasználói felületen.

JavaScriptben az eseménykezelőket függvényként kell megvalósítani, a beolvasás és a kiírás pedig a DOM-on keresztül történik, ahogy azt az előző fejezetben megismerhettük.

```
function esemenykezelo() {
  // beolvasás
  // feldolgozás
  // kiírás
}
```

5.2 Események

A felhasználó tevékenysége **sokféle eseményt** ↪ válthat ki az oldalon. Tipikus események:

- `click` : egérkattintás
- `mousemove` : egérmozgatás
- `mousedown` : egér gombjának lenyomása
- `mouseup` : egér gombjának felenegedése
- `keydown` : billentyűzet gombjának lenyomása
- `keyup` : billentyűzet gombjának felengedése
- `keypress` : billentyűzet gombjának megnyomása
- `submit` : űrlap elküldése
- `scroll` : görgetés az oldalon

5.3 Eseménykezelő függvények regisztrálása

Az események mindig valamelyik DOM elemhez kapcsolódnak. Ha a felhasználó megnyom egy gombot, akkor az a gomb fogja a `click` eseményt jelezni; ha gépel egy szöveges beviteli mezőben, akkor az az `input` mező fogja a `keypress` eseményeket jelezni; ha görgeti az oldalt, akkor a `window` objektum jelzi ezt egy `scroll` esemény dobásával.

Egy DOM elemen bekövetkező eseményre a DOM elem `addEventListener` metódusával lehet feliratkozni. Másképpen: ezzel a metódussal lehet egy eseménykezelő függvényt hozzákapcsolni a DOM elemen jelentkező eseményhez. Ha az esemény bekövetkezik a DOM elemen, akkor az eseményhez kapcsolt eseménykezelő függvény meghívódik, és a benne lévő program lefut. Egy eseménykezelő függvényt eltávolítani pedig a `removeEventListener` metódussal lehet.

```
// általánosan
elem.addEventListener(esemény_típusa, eseménykezelő_függvény);
elem.removeEventListener(esemény_típusa, eseménykezelő_függvény);

// például
gomb.addEventListener("click", kattintas);
gomb.removeEventListener("click", kattintas);

function kattintas() {
  // mi történjen kattintáskor
}
```

Az `eseménykezelő_függvény` paraméter függvényhivatkozást tartalmaz, azaz csak a függvény nevét kell oda beírni, meghívni (`eseménykezelő_függvény()`) nem szabad. Az eseménykezelő eltávolításakor ugyanazt a függvényhivatkozást kell megadni a `removeEventListener` függvénynek, mint amit regisztráláskor megadtunk.

Lehetőség van a függvényt helyben is definiálni:

```
elem.addEventListener(esemény_típusa, function () {
    // eseménykezelő kód
});
```

Apróbetűs

Egy elem egy eseményéhez több eseménykezelő függvény is kapcsolható.

```
gomb.addEventListener("click", kattintas1);
gomb.addEventListener("click", kattintas2);
```

Példa

Kérjük be a felhasználó nevét, majd üdvözöljük őt!

```
<input id="nev">
<button id="gomb">Üdvözöl</button>
<span id="kimenet"></span>

<script>
  $("#gomb").addEventListener("click", kattintas);
  function kattintas() {
    // beolvasás
    const nev = $("#nev").value;
    // feldolgozás
    const udvozles = `Hello ${nev}!`;
    // kiírás
    $("#kimenet").innerHTML = udvozles;
  }
</script>
```

Apróbetűs

Az eseménykezelő függvényt történeti okok miatt sokféleképpen lehet regisztrálni. Ezek közül az `addEventListener` a szabványos és a legrugalmasabb megoldás. Érdemes azonban megismerkedni az egyik legelső megoldással, amely HTML attribútumon (`on*`) keresztül rendelte hozzá az eseménykezelő függvényt az adott elemen bekövetkező eseményhez:

```
<elem ontipus="esemenykezelo()">
<!-- Például -->
<button onclick="kattintas()">Üdvözöl</button>
```

Ez a fajta jelölésmód újra teret nyer a modern kliensoldali keretrendszerekben.

5.4 Az eseményobjektum

Az esemény aktuális bekövetkezéséhez tartozó adatokat az ún. *eseményobjektum* tartalmazza. Ebben olyan információk szerepelnek többek között, mint pl. az **aktuálisan lenyomott billentyű ↪** kódja (`key` , `code`) vagy az **egérkurzor helyzete ↪** (`clientX` , `clientY` , `screenX` , `screenY`) a képernyőn.

Az eseményobjektumot az eseménykezelő függvény első paramétereként automatikusan rendelkezésünkre bocsátja a böngésző. Az eseményobjektum aktuális tartalmáról legegyszerűbben

úgy győződhetünk meg, ha kiíratjuk konzolra a tartalmát.

```
function esemenykezelo(e) {
  console.log(e);
}
```

Példa

Rajzoljunk ki egy csillagot a képernyőnek azon pontjára, ahova kattintottunk!

```
<style>
  .csillag {
    position: fixed;
    list-style-type: none;
  }
</style>
<ul id="csillagok"></ul>
<script>
  document.addEventListener("click", kattintas);
  function kattintas(e) {
    // beolvasás
    const x = e.clientX;
    const y = e.clientY;
    // feldolgozás
    const csillag = `<li class="csillag" style="top: ${y}px; left:
${x}px;">*</li>`;
    // kiírás
    $("#csillagok").innerHTML += csillag;
  }
</script>
```

5.5 Események buborékolása és delegálása

Egy esemény bekövetkezte mindig egy adott DOM objektumhoz kapcsolódik. Ezt nevezzük az esemény *forrásobjektum*ának. Azonban az eseményt nemcsak ez az objektum jelzi, hanem annak szülője, majd annak szülője, szép sorban egészen a legfelső szintig a `document` objektumig. Ezt nevezzük az *esemény buborékolásá*nak.

Ez azt is jelenti, hogy egy eseményt nemcsak azon a szinten lehet kezelni, ahol az bekövetkezik, hanem fölötte tetszőleges szinten. Az eseményobjektumon `target` tulajdonságán keresztül pedig le lehet kérdezni az esemény forrásobjektumát. Ha egy eseményt felsőbb szinten kezelünk, de az eseménykezelőben a forrásobjektummal dolgozunk, akkor azt az *esemény delegálásá*nak hívjuk.

A delegált eseménykezelés azokban az esetekben hasznos, amikor sok hasonló vagy dinamikusan beszúrt elemhez kellene eseménykezelőket társítanunk. Ekkor megkeressük az érintett elemek legközelebbi közös ősét, és vagy ahhoz, vagy egy tetszőlegesen fölötte lévő elemhez kötjük az eseménykezelőt. Ezzel a sok eseménykezelő helyett eggyel megoldhatjuk a feladatot, ráadásul a dinamikusan hozzáadott újabb elemekre is automatikusan érvényes lesz az így létrehozott logika.

A felsőbb szintre sokféle forrásból érkezhet esemény, a `target` objektumot érdemes valamilyen módon megszűrni, például megnézni a `matches` metódussal, hogy egy adott CSS szelektor illeszkedik-e rá.

Példa

Egy listaelemre kattintva váltogassuk annak stílusosztályát!

```
<style>
  .kesz:before {
    content: "✓ ";
  }
</style>
<ul class="lista">
  <li>első</li>
  <li>második</li>
  <li>harmadik</li>
</ul>
<script>
  $("ul.lista").addEventListener("click", listaKattintas);
  function listaKattintas(e) {
    if (e.target.matches("li")) {
      // beolvasás
      const li = e.target;
      // kiírás
      li.classList.toggle("kesz");
    }
  }
</script>
```

Apróbetűs

Speciális esetekben nehéz lehet a delegálás megoldása. Például, ha az előző példában a listaelemeken belül egy `<span>` elem is lenne, akkor az `e.target` objektum arra mutatna, és az elágazásunk feltétele nem teljesülne. Általánosan megfogalmazva: előfordulhat, hogy a forrásobjektum és a regisztrált szint közötti szinten lévő objektumhoz szeretnénk az eseménykezelőt rendelni. Ennek elősegítésére bevezethetünk egy `delegal` segédfüggvényt, amely végigmegy a forrásobjektumtól a kezelt szintig, megkeresve azt az első elemet, akire a megadott CSS szelektor illeszkedik. Az előző példa kódja így nézne ki vele:

```
function delegal(szulo, tipus, szelektor, fuggveny) {

  function delegaltFuggveny(e) {
    if (e.target.matches(`${szelektor},${szelektor} *`)) {
      let celpont = e.target;
      while (!celpont.matches(szelektor)) {
        celpont = celpont.parentNode;
      }
      e.valodiCelpont = celpont;
      return fuggveny.call(celpont, e);
    }
  }

  szulo.addEventListener(tipus, delegaltFuggveny);
}

delegal($("ul.lista"), "click", "li", listaKattintas);
function listaKattintas(e) {
    // beolvasás: this === a delegált objektum
    const li = this;
    // kiírás
    li.classList.toggle("kesz");
}
```

5.6 jQuery

Apróbetűs

A jQuery egy kliensoldali keretrendszer. Segítségével kényelmes programozási interfészen keresztül tudjuk a HTML felületet programozni.

Telepítés

Szkriptjeink elé a következő `<script>` elemeket kell beszúrni:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"
        integrity="sha256-
FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
        crossorigin="anonymous"></script>
```

Használat

A jQuery egy `jQuery` vagy `$` függvényt bocsát rendelkezésünkre, mindent ezen keresztül tudunk megtenni. A `$` függvény hármas szerepet tölt be:

- `$(szelektor)` : kiválasztja a szelektornak megfelelő elemeket (ez hasonlít az általunk bevezetett `$` függvényhez);
- `$(html_szöveg)` : létrehozza a memóriában a paraméterként megadott HTML szövegnek megfelelő elemeket;
- `$(függvény_hivatkozás)` : az oldal betöltése után lefuttatja a paraméterként megadott függvényt, egyfajta inicializálásként.

A jQuery használata során két alapvető műveletünk van:

1. Kiválasztjuk a megfelelő elemeket (`$(szelektor)`)
2. A kiválasztott elemeken végrehajtjuk a kívánt műveleteket (`kiválasztott_elemek.művelet()`).

Példa

Tüntessük el az összes paragrafust!

```
$("p").hide();
```

Fontosabb műveletek

- Bejárás

```
// Gyerekekre lépés
$("#valami").children();
$("#valami").find("li");

// Szülőkre, ősökre lépés
$("#valami").parent();
$("#valami").closest("form");

// Testvérek kiválasztása
$("#valami").siblings();
$("#valami").next();
$("#valami").nextAll();
$("#valami").prev();
$("#valami").prevAll();
```

- Elemek attribútumai, tartalma

```
// Attribútumok kezelése
$("#valami").attr("attr");
$("#valami").attr("attr", "érték");

// innerHTML kezelése
$("#valami").html();
$("#valami").html("html");

// Űrlapelemek kezelése
$("#valami").val();
$("#valami").val("érték");
```

- Stílusmanipuláció

```
// Style attribútum programozása
$("#valami").css("display", "none");
$("#valami").css({
  color: "blue",
  "background-color": "yellow"
});
$("#valami").css("width")

// Stílusosztályok kezelése
$("#valami").addClass("fontos");
$("#valami").removeClass("fontos");
$("#valami").toggleClass("fontos");
$("#valami").hasClass("fontos")
```

```
// Nevesített stílusértékek kezelése
$("#valami").height(100);

// Animációk
$("div").hide();
$("div").fadeIn();
```

- Eseménykezelés

```
// Eseménykezelés
$("#valami").on("click", kattintas);

// Delegálás
$("#valami").on("click", "li", kattintas);
function kattintas(e) {
    // this === a delegált elem
}
```

5.7 Feladatok

1. **Nagyfeladat**

   Adott nevek listája. Húzd át azokat a neveket, amelyekre egyszerre mindkét egérgombbal kattintottunk. Az egéresemények közül a `mousedown` eseménynél elérhető egy `buttons` tulajdonság, amelyen keresztül lekérdezhető, hogy mely egérgombok voltak egyidejűleg lenyomva. Ügyelni kell arra is, hogy a jobb egérgomb lenyomásakor ne jelenjen meg a helyi menü. Ehhez a `contextmenu` esemény alapértelmezett műveletét kell letiltani.

2. **Nagyfeladat**

   Adott egy táblázat. Írjuk ki a táblázat alá a kattintott cella sor-oszlop információját! Egy táblázatcella objektum `cellIndex` metódusa megadja, hogy a cella hányadik a sorban. Egy sor objektum `sectionRowIndex` metódusa pedig megadja, hogy a sor hányadik a táblázatban. A cella `parentNode` tulajdonsága adja meg a sorát.

Letöltés

- ⤓ A fejezethez tartozó nagyfeladatok megoldásai ↪
- ⤓ Az aknakeresős nagyfeladat teljes megoldása ↪

6 A KLIENSOLDALI ALKALMAZÁSFEJLESZTÉS ALAPELVEI

Ebben a fejezetben a böngészőben futó eseményvezérelt grafikus alkalmazások készítéséhez szükséges magasabb szintű alapelvekkel ismerkedünk meg.

Az előző fejezetekben láthattuk, hogy milyen vezérlési és adatszerkezetekkel lehet JavaScriptben dolgozni, egyszerűbb feladatokat, programozási tételeket megoldani; hogy hogyan lehet a böngészőbe betöltött HTML elemeket JavaScriptből elérni és programozni a DOM-on keresztül; végül azt is, hogy hogyan tud a felhasználó kapcsolatba lépni az alkalmazással az események és eseménykezelők segítségével. Ezek az ismeretek szolgálnak a kliensoldali alkalmazásfejlesztés építőköveinek, amelyekből nagyobb alkalmazások rakhatók össze. A fejlesztés folyamatát ugyanakkor számos *magasabb szintű elv* segíti. Ebben a fejezetben ezeket az elveket tekintjük át, és adunk ajánlást a kliensoldali webes alkalmazások kódszervezésére.

6.1 Alkalmazásfejlesztési alapelvek

Ugyan a DOM egy kiváló programozási interfészként szolgál a JavaScript kód és a felületi elemek között, általában igaz, hogy próbáljuk meg a feldolgozási logika és a felületi logika érintkezési pontját minél kisebbre venni. Másképpen szólva: *az adatok feldolgozását válasszuk el az adatok megjelenítésétől*. Így az alkalmazásunk kódja két fő részre bomlik.

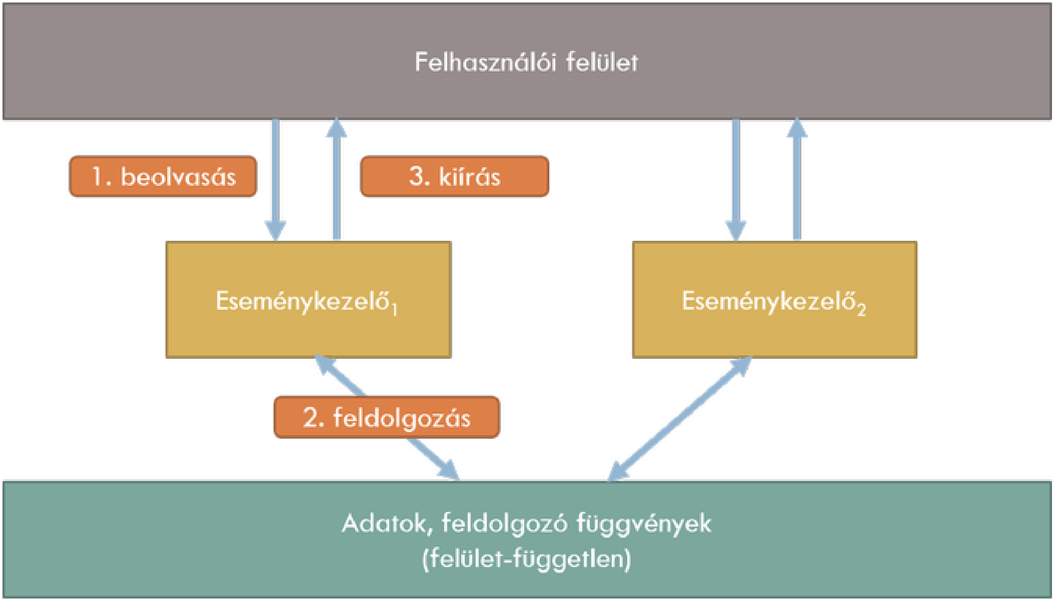
Az egyik részt az alkalmazás lényegi, logikai része alkotja (üzleti logika). Ez független lesz a felülettől, és pusztán nyelvi elemek használatával megvalósítható.

Az alkalmazás másik részét a felület kezelése adja, azaz a beolvasás és a kiírás implementálása. Ebben a részben jelennek meg a DOM műveletek, amelyek vagy kiolvassák, vagy beállítják a megfelelő DOM objektum megfelelő adattulajdonságát. Az egyes résztevékenységeket általában felhasználói aktivitás váltja ki: egy gombnyomás, gépelés, egérmozgás. Az ennek hatására lefutó eseménykezelő függvények azok a miniprogramok, amelyek a beolvasás-feldolgozás-kiírás hármasát megvalósítják.

Végül megjelenhetnek olyan segédfüggvények, amelyek a konkrét feladattól függetlenül egy általános részfeladat megoldását végzik el.

Az alkalmazásunkhoz tartozó kód tehát alapvetően három részből áll, ahogy azt az alábbi ábra is mutatja:

- adatok deklarálásáért és azok feldolgozásáért felelős kódrészekből;
- a felület kezelését végző kódrészekből; valamint
- segédfüggvényekből.



Egy JavaScript program működésének sematikus ábrája

6.2 Az alkalmazás állapota

Az alkalmazás állapotát azok az adatok képviselik, amelyek szükségesek az alkalmazás mindenkori működtetéséhez. Ezek az adatok jelentik az alkalmazás magját, minden további logika e köré épül, ennek az állapotnak a megjelenítését, változtatását szolgálja. Minden alkalmazás célja, hogy egy kezdeti állapotból egy végállapotba jusson el. Ez jelenti az alkalmazás sikeres működését.

Az adatokat nyelvi elemekkel, esetünkben a JavaScript adatszerkezeteivel (egyszerű típusok, objektum, tömb) írjuk le. Ehhez használhatunk egyszerű globális változókat, vagy ezeket egyetlen objektumba is foglalhatjuk. Az egyszerűség kedvéért ez a tananyag több globális változóval operál. Az adatok leírása független a felületi elemektől, így HTML- vagy DOM-specifikus rész nem jelenhet meg benne.

Példa

Számkitalálós játék: a gép gondolt egy számra 1 és 100 között, találjuk ki!

Biztosan tárolnunk kell a kitalálandó számot. Emellett jó lenne azt is tudni, hogy kitaláltuk-e már a számot. Továbbá korábbi tippjeinket is meg kellene jeleníteni, így azokat is tárolnunk kell. Ezekkel már a játék teljes mértékben működtethető (megjeleníthető). Opcionálisan tárolhatjuk az aktuális tippet is.

```
// globális változókkal
let kitalalandoSzam = 42;
let vege = false;
const tippek = [50, 25, 38, 44];

// objektumba foglalva
const allapot = {
  kitalalandoSzam: 42,
  vege: false,
  tippek: [50, 25, 38, 44]
}
```

6.3 Az állapot változtatása

Az alkalmazás állapota folyamatosan változik az alkalmazás működése során. Ezeket az ún. *állapot-átmenetek*et tiszta nyelvi elemekkel idézzük elő, így ez a rész is mentes a felület-specifikus műveletektől. Egy klasszikus programban ezek a feldolgozó függvények, amelyek egy adott bemenethez előállítják a kimenetet. A mi esetünkben a feldolgozó függvények megváltoztatják az alkalmazás állapotát a bemeneti adatoknak megfelelően, illetve kinyerik belőle a kimeneti adatokat. A feldolgozó logika helyet kaphat közvetlenül az eseménykezelő függvényen belül is, de szerencsésebb a jól megfogható funkcióval rendelkező műveleteket külön függvényekbe szervezni.

Példa

A számkitalálós példában az állapot-átmenetet a felhasználó tippelése indukálja. A felületről beolvasott értéket be kell szúrni a `tippek` tömbbe, össze kell hasonlítani a `kitalalandoSzam` értékével, és ennek megfelelően módosítani kell a `vege` állapotot.

```
function tipp(tippeltSzam) {
  tippek.push(tippeltSzam);
  vege = (tippeltSzam === kitalalandoSzam);
}
```

6.4 Segédfüggvények

A segédfüggvények nem járulnak hozzá az alkalmazás lényegi logikájához, hanem annak működését segítik jól elkülöníthető, általános és újrahasznosítható funkciók függvényekbe zárásával. Ilyen segédfüggvény pl. a `$` függvény.

Példa

A számkitalálós játék esetében a játék kezdőállapotának beállításakor szükséges egy véletlen egész szám generálása. Mivel JavaScriptben a `Math.random` függvény egy 0 és 1 közötti lebegőpontos számot állít elő, így készíthetünk egy segédfüggvényt, ami ennek felhasználásával egy `min` és `max` közé eső egész számot állít elő (ez a segédfüggvény korábbi fejezetben feladatként szerepelt).

```
function veletlenEgesz(min, max) {
  const veletlen = Math.random();
  const tartomany = max - min + 1;
  return Math.trunc(veletlen * tartomany) + min;
}
```

6.5 Eseménykezelő függvények

Az eseménykezelő függvények jelentik az *alkalmazás belépési pontjai*t. Definiálásukkor mindig gondoljuk át, hogy melyik elem milyen eseményére szeretnénk reagálni. Amennyiben több hasonló elemhez szeretnénk eseménykezelőt rendelni, használjunk delegálást. Az eseménykezelők regisztrálását vagy a hozzájuk tartozó függvényeknél végezzük el, vagy külön, egy helyen csoportosítjuk őket.

Felépítésük megfelel a klasszikus beolvasás-feldolgozás-kiírás hármasának.

- *beolvasás*: a DOM-ból kiolvassa az adatokat és lokális változókba eltárolja;
- *feldolgozás*: vagy közvetlenül módosítja az állapotot, vagy egy feldolgozó függvényt hív meg, ami az alkalmazás állapotával dolgozik; ha van visszatérési érték, akkor azt lokális változóban tárolja;
- *kiírás*: a kimeneti adatokat, amik lehetnek állapotbeli adatok vagy lokális változók, DOM műveletek segítségével megjeleníti a felületen.

Példa

Számkitalálós játékunkban a felhasználó beír egy számot egy szöveges beviteli mezőbe, majd megnyom egy gombot. A gomb megnyomására kiolvassuk a szöveges beviteli mező értékét, majd meghívjuk a `tipp` feldolgozó függvényt, végül megjelenítjük az eddigi tippjeinket azzal az információval együtt, hogy az a kitalálandó számhoz hogyan viszonyul (kisebb, nagyobb, egyenlő). Vége esetén nem engedünk többet tippelni, letiltjuk a gombot.

```
<input id="tipp">
<button id="tippGomb">Tipp!</button>
<ul id="tippek"></ul>

<script>
  $("#tippGomb").addEventListener("click", tippeles);
  function tippeles(e) {
    // beolvasás
    const tippeltSzam = parseInt($("#tipp").value);
    // feldolgozás
    tipp(tippeltSzam);
    // kiírás
    $("#tippek").innerHTML = tippek.map(szam =>
      `<li>${szam} (${hasonlit(szam, kitalalandoSzam)})</li>`
    ).join("");
    $("#tippGomb").disabled = vege;
  }
  function hasonlit(szam, kitalalandoSzam) {
    if (szam < kitalalandoSzam) return "nagyobb";
    if (szam > kitalalandoSzam) return "kisebb";
    return "egyenlő";
  }
</script>
```

6.6 HTML generáló függvények

A kimenet előállításakor vagy egy DOM elem tulajdonságát módosítjuk (pl. a `disabled` tulajdonság a fenti példában), vagy HTML szöveget illesztünk be egy nyitó- és záróelem közé (ld. a listaelemek generálását fent). Ez utóbbi esetben gyakran előfordul nagyobb mennyiségű HTML

szöveg generálása, amit nem helyben, hanem külön függvényben végzünk el. A HTML generáló függvény megkapja paraméterül a megjelenítendő állapotrészt, majd az előállított HTML szöveggel tér vissza. A generáló függvények akár más generáló függvényeket hívhatnak.

Példa

Az előző fejezetben a lista generálását külön függvénybe szervezhetjük:

```
function genLista(tippek, kitalalandoSzam) {
  return tippek.map(szam =>
    `<li>${szam} (${hasonlit(szam, kitalalandoSzam)})</li>`
  ).join("");
}
```

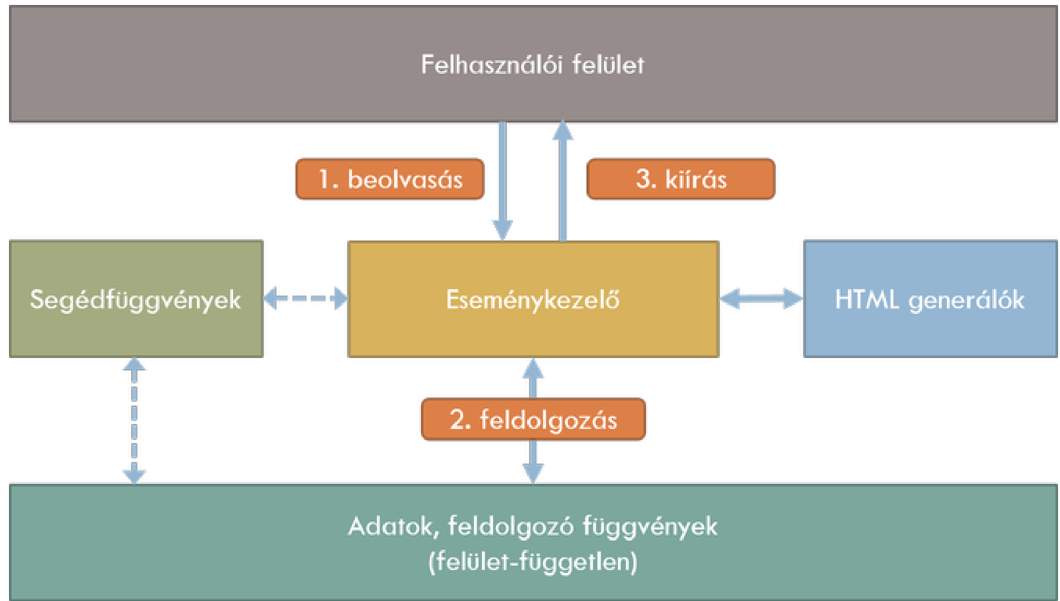
Ezzel áttekinthetőbb lesz az eseménykezelőnk:

```
function tippeles(e) {
  // beolvasás
  const tippeltSzam = parseInt($("#tipp").value);
  // feldolgozás
  tipp(tippeltSzam);
  // kiírás
  $("#tippek").innerHTML = genLista(tippek, kitalalandoSzam);
  $("#tippGomb").disabled = vege;
}
```

6.7 Kódszervezés

A HTML generáló függvények használata nagyon gyakori az implementáláskor. Velük együtt a következő csoportokba sorolhatók az egyes kódrészletek:

- állapot-leíró és állapot-átmenetet végző függvények (adatok és feldolgozásuk),
- eseménykezelő függvények (az alkalmazás belépési pontjai),
- HTML generáló függvények (speciális segédfüggvények a kiíráshoz),
- feladatfüggetlen segédfüggvények.



Kódszervezés JavaScript-ben

Példa

Számkitalálós játék

Számkitalálós játékunk végső kódja néhány apró változtatással így néz ki:

- hibaellenőrzés beolvasáskor: hiba esetén piros keret rajzolása és az eseménykezelő futásának megszakítása;
- kiírás: tippeléskor a keret színének visszaállítása, a beviteli mező törlése, kurzor visszahelyezése; találatkor a beviteli mező is elérhetetlenné válik.

```
<h1>Számkitalálós játék</h1>
<p>Gondoltam egy számra 1 és 100 között. Találd ki!</p>
<input id="tipp">
<button id="tippGomb">Tipp!</button>
<ul id="tippek"></ul>

<script>
  /////////////////////
  // SEGÉDFÜGGVÉNYEK //
  /////////////////////

  function $(szelektor) {
    return document.querySelector(szelektor);
  }
  function veletlenEgesz(min, max) {
    const veletlen = Math.random();
    const tartomany = max - min + 1;
    return Math.trunc(veletlen * tartomany) + min;
  }
```

```
////////////////////
// ÁLLAPOTTÉR      //
////////////////////

let kitalalandoSzam = veletlenEgesz(1, 100);
let vege = false;
const tippek = [];

function tipp(tippeltSzam) {
  tippek.push(tippeltSzam);
  vege = tippeltSzam === kitalalandoSzam;
}

////////////////////
// ESEMÉNYKEZELŐK  //
////////////////////

$("#tippGomb").addEventListener("click", tippeles);
function tippeles(e) {
  // beolvasás
  const tippeltSzam = parseInt($("#tipp").value);
  if (isNaN(tippeltSzam)) {
    $("#tipp").style.borderColor = "red";
    return;
  }
  // feldolgozás
  tipp(tippeltSzam);
  // kiírás
  $("#tippek").innerHTML = genLista(tippek, kitalalandoSzam);
  $("#tippGomb").disabled = vege;
  $("#tipp").disabled = vege;
  $("#tipp").value = "";
  $("#tipp").focus();
  $("#tipp").style.borderColor = "";
}

////////////////////
// HTML GENERÁLÓK  //
////////////////////

function genLista(tippek, kitalalandoSzam) {
  return tippek.map(szam =>
    `<li>${szam} (${hasonlit(szam, kitalalandoSzam)})</li>`
  ).join("");
}
function hasonlit(szam, kitalalandoSzam) {
  if (szam < kitalalandoSzam) return "nagyobb";
  if (szam > kitalalandoSzam) return "kisebb";
  return "egyenlő";
```

```
  }
</script>
```

6.8 Feladatok

1. **Nagyfeladat**

   Készítsd el az Aknakereső játékot a fenti felosztás figyelembe vételével!

   - **Felület**
     - A felhasználói felületen lehessen megadni a játéktábla méreteit és az aknák számát. Egy gombra kattintva megjelenik a megfelelő méretű játéktér egy táblázatban, minden cellában egy-egy gombbal.
     - Bal kattintásra felfedjük az adott cellában lévő értéket. Ha ez üres mező, akkor amíg üres mezőt találunk körülötte, addig azokat is felfedjük.
     - Jobb kattintásra zászlót helyezünk el az adott cellában.
     - A bal és jobb egérgomb egyidejű lenyomására pedig az adott cella körüli 8 mezőt fedjük fel.
     - Ha a felfedés során bombára bukkanunk, akkor veszítettünk, az egész pályát felfedjük és vége a játéknak.
     - Ha minden elemet felfedtünk és csak az aknák maradtak, akkor nyertünk és vége a játéknak.
     - Ügyelni kell arra, hogy bizonyos műveletek csak bizonyos állapotú cellán érvényesek. Pl. egy felfedett cellát még egyszer nem tudunk felfedni.
   - **Állapottér** Az állapottér tartalmazza a következőket:
     - a játéktábla adatszerkezetét (mátrix),
     - a tábla méreteit,
     - az aknák számát,
     - a zászlók számát,
     - a játékállapotot (kezd, játék, nyer, veszít).
   - **Állapot-átmenetek** Az állapotteret a következő műveletek változtathatják:
     - kezdőállapotba hozás
     - pályagenerálás
     - felfedés
     - üres cellák felfedése egy adott pontból
   - **Eseménykezelők**
     - Start gomb megnyomása
     - Cellában lévő gombon bal kattintás
     - Cellában lévő gombon jobb kattintás
     - Cellában lévő gombon kétgombos kattintás

- **HTML generálók**
  - Pálya kirajzolása az állapottér alapján
- **Segédfüggvények**
  - `$`
  - `xyKoord`
  - véletlenszám generálás
  - mátrix létrehozása
  - mátrixbeli szomszédos elemek növelése

6.9 A kimenet mint állapot-leképezés

A kimenet-generálás egy speciális fajtája, amikor minden állapotváltozáskor a teljes felületet újrageneráljuk az állapot szerint. Ez a koncepció jól mutatja, hogy a felület a mindenkori állapottér HTML elemekre való leképezése valójában. Ezért fontos az állapotteret jól definiálni, mert az alapján bármikor elő- vagy visszaállítható a felület. Nem a felületi elemek vezérlik az adatokat, hanem az adatok jelennek meg a felületi elemekben.

Gyakorlati megvalósításképpen egy `kirajzol` nevű függvényt érdemes létrehozni, ami az állapot alapján egy cél DOM elembe generálja a HTML szöveget. Innentől kezdve minden eseménykezelő függvény a beolvasás és feldolgozás után a `kirajzol` függvényt hívja meg a felület konzisztensen tartására.

```
let cim = "Hello világ!";
function kirajzol() {
  $("#celELem").innerHTML = genAlkalmazas(cim);
}
function genAlkalmazas(cim) {
  return `
    <div>
      ${genCim(cim)}
    </div>
  `;
}
function genCim(cim) {
  return `<h1>${cim}</h1>`;
}
```

A fenti elmélet a gyakorlatban több hátulütővel is jár. A teljes felület újragenerálása komplex alkalmazás esetén egyrészt lassú is lehet, másrészt az űrlapelemek – mindig kicserélődve – elveszíthetik a fókuszt.

Apróbetűs

A hátulütők elkerülésére a gyakorlatban olyan függvénykönyvtárakat használnak, amelyek nem az egész DOM-ot cserélik ki, hanem okosan összehasonlítják a meglévő DOM elemeket az újonnan beszúrandókkal, és csak a szükséges változásokat szúrják be a DOM-ba. Ezzel a megoldással a felület kezelése nagyon gyorssá válhat, és a fejlesztőnek sem kell az egyes DOM elemeket módosítania, neki elég a teljes felületet újrarajzoltatnia. Ezeket a függvénykönyvtárakat *DOM összehasonlító könyvtárak*nak szokták nevezni.

Apróbetűs

Komponensek

Az előző fejezetbeli HTML generáló függvények a felület egy-egy jól meghatározott részének kirajzolásáért feleltek. Ezt a gondolatot tovább víve, ezeknek a függvényeknek a felelősségi körét tovább növelhetjük: nemcsak az adott felületi rész kirajzolását, hanem az összes vele kapcsolatos kimeneti és bemeneti művelet kezelését is ők végezhetik. Ennek megfelelően a felhasználói felületet funkcionálisan egységes részekre bonthatjuk, és minden egyes rész kezeléséhez egy JavaScript objektumot rendelünk. Az így kialakult objektumokat és a hozzá tartozó felületi részeket *komponensek*nek nevezzük. Ahogy a HTML generáló függvények meghívhattak más HTML generáló függvényeket, úgy egy komponens is több komponensből állhat. Végső soron az alkalmazás is egy nagy komponens, amely kisebb komponensekből tevődik össze, amelyek újabb komponensekre bomlanak, és így tovább.

A komponensek határait a fejlesztő dönti el. Lehetnek ezek nagyon kis felületi egységek, mint például egy gomb vagy valamilyen beviteli elem, de akár bonyolultabb HTML elemeket is egységbe foglalhat.

Példa

A számkitalálós játékot is átírhatjuk ennek megfelelően. Az állapotot ebben az esetben egy globális objektumba csoportosítottuk, és az állapot-átmeneti függvényeket is ennek az objektumnak a részévé tettük. Így az állapotkezelés ennek az objektumnak a felelőssége.

A kiindulási HTML szerkezet ebben az esetben egy tartalmazó `div`-ből áll. Az alkalmazásban három komponenst különböztetünk meg. Az `Alkalmazas` az egész alkalmazás kiírásáért és újrarajzolásáért felel, ő a legfelső szintű komponens. Két komponenst generál: a `Tipp` komponens a beviteli mezőkért felel, és a hozzájuk kapcsolódó eseménykezelőkért; a `Lista` komponens pedig a lista megjelenítéséért.

A következő változások történtek még ezeken kívül:

- az eseménykezelő regisztrálása ebben az esetben az ún. inline módszerrel történt. Ekkor HTML attribútumként adhatjuk meg az eseménykezelőt,

explicíten átadva paraméterül az `event` nevű eseményobjektumot.

- globális állapot: a komponensek rálátnak az állapotra, így a kirajzol függvények is közvetlenül hivatkoznak rá.
- fókusz: az állandó újrarajzolás deklaratív volta miatt – azaz azzal, hogy az adatok határozzák meg a felület kirajzolását – a fókusz kezelését elveszítettük. Tulajdonképpen a kiírás során nem tudunk egyesével a generált elemekbe nyúlni, csak az állapottéren keresztül.

```
<div id="alkalmazas"></div>

<script>
  ////////////////////
  // SEGÉDFÜGGVÉNYEK //
  ////////////////////

  function $(szelektor) {
    return document.querySelector(szelektor);
  }

  function veletlenEgesz(min, max) {
    const veletlen = Math.random();
    const tartomany = max - min + 1;
    return Math.trunc(veletlen * tartomany) + min;
  }

  ////////////////////
  // ÁLLAPOTTÉR      //
  ////////////////////

  const allapot = {
    kitalalandoSzam: veletlenEgesz(1, 100),
    vege: false,
    tippek: [],

    tipp: function (tippeltSzam) {
      allapot.tippek.push(tippeltSzam);
      allapot.vege = tippeltSzam === allapot.kitalalandoSzam;
    }
  }

  ////////////////////
  // KOMPONENSEK     //
  ////////////////////

  const Alkalmazas = {
    kirajzol: function () {
      return `
        <h1>Számkitalálós játék</h1>
```

```
        <p>Gondoltam egy számra 1 és 100 között. Találd ki!
</p>
        ${!allapot.vege
          ? Tipp.kirajzol()
          : "<p>Gratulálunk, kitaláltad!</p>"
        }
        ${Lista.kirajzol()}
      `;
    },
    ujrarajzol: function () {
      $("#alkalmazas").innerHTML = Alkalmazas.kirajzol();
    }
  };

  const Tipp = {
    ertek: "",
    hibas: false,
    tippeles: function (e) {
      // beolvasás
      Tipp.hibas = false;
      const tippeltSzam = parseInt(Tipp.ertek);
      if (isNaN(tippeltSzam)) {
        Tipp.hibas = true;
        Alkalmazas.ujrarajzol();
        return;
      }
      // feldolgozás
      allapot.tipp(tippeltSzam);
      // kiírás
      Tipp.ertek = "";
      Alkalmazas.ujrarajzol();
    },
    valtozas: function (e) {
      // beolvasás és kiírás
      Tipp.ertek = e.target.value;
    },
    kirajzol: function () {
      return `
        <input value="${Tipp.ertek}"
          ${Tipp.hibas ? `style="border-color: red"` : ""}
          oninput="Tipp.valtozas(event)"
        >
        <button onclick="Tipp.tippeles(event)">Tipp!</button>
      `;
    }
  }

  const Lista = {
    kirajzol: function () {
      return `
```

```
        <ul>
          ${allapot.tippek.map(szam =>
            `<li>${szam} (${Lista.hasonlit(szam,
allapot.kitalalandoSzam)})</li>`
          ).join("")}
        </ul>
      `;
    },
    hasonlit: function (szam, kitalalandoSzam) {
      if (szam < kitalalandoSzam) return "nagyobb";
      if (szam > kitalalandoSzam) return "kisebb";
      return "egyenlő";
    }
  }

  Alkalmazas.ujrarajzol();
</script>
```

7 RAJZOLÁS JAVASCRIPT SEGÍTSÉGÉVEL

Az eddigiekben megismerkedtünk a HTML elemek programozásával, azok eseményeinek kezelésével. Ebben a fejezetben a programozott grafikák készítse, rajzolással készített szimulációk és játékok kerülnek a fókuszba. Célunk, hogy a fejezet végére elkészítsünk egy egyszerűbb, rajzvászonnal működő játékot, a közismert Flappy Bird játék egy változatát.

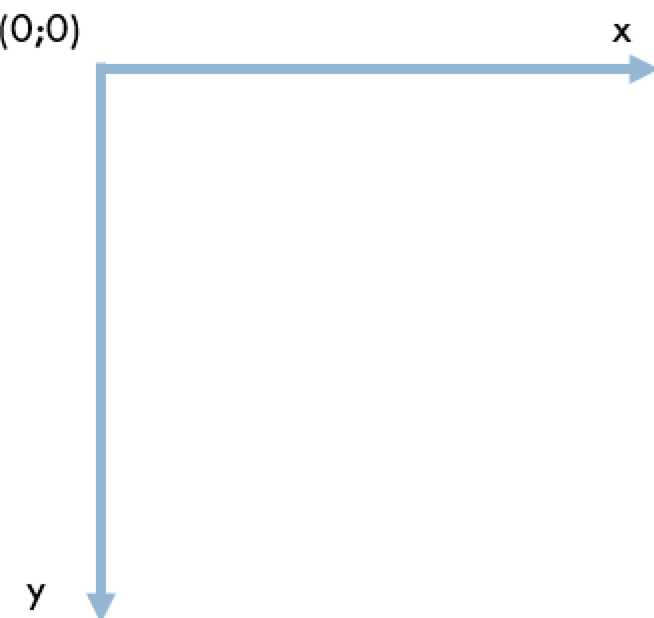
7.1 Rajzolás vászonra

A HTML 5.0-s verziójával került bevezetésre a `canvas` elem, ami egy rajzvászon leírását teszi lehetővé. Önmagában egy ilyen vászon kevés dologra alkalmas, de a megfelelő programmal könnyen életre kelthetjük.

A `canvas` ↪ elemre JavaScript segítségével programozottan készíthetünk ábrákat. Ez a megközelítés némileg hasonlít a Teknőcgrafikához abban, hogy utasítások és egy képzeletbeli "toll" segítségével készül az ábra. A `canvas` elemhez tartozó DOM objektum több, úgynevezett kontextussal rendelkezik. Ezek a kontextusok lehetővé teszik különböző típusú grafikák készítését, például OpenGL technológiás 3D grafikákat, vagy esetünkben egyszerű két dimenziós ábrák készítését. Ahhoz, hogy ezeket a kontextusokat használni tudjuk, el kell érnünk a `canvas` tag-hez tartozó DOM elemet, és annak el kell kérni a megfelelő kontextusát.

```
const vaszon = $("canvas");
const rajz = vaszon.getContext("2d");
```

A továbbiakban minden rajzoló műveletet ezzel a **kontextussal** ↪ (a kódban `rajz` változó) fogunk végezni. A számítógépes grafikában leggyakrabban valamilyen koordináta-rendszerben gondolkozunk. Jellemzően ezen koordinátarendszerek `(0;0)` pontja a bal felső sarokban található, és az `x` tengely jobbra, míg az `y` tengely lefelé növekszik, egy képpont (pixel) felosztásúak. Ezek a szabályok a `canvas` alapú grafikára is igazak.



Koordinátarendszer a számítógépen

7.1.1 Egyszerű alakzatok rajzolása vászonra

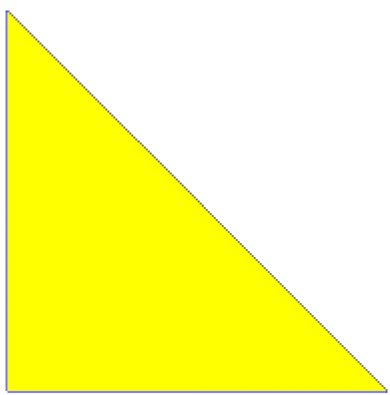
Ahhoz, hogy a koordinátarendszerünkben egyszerűen tájékozódhassunk tudnunk kell a rajzoló terület méretét. Egy `canvas` elem alapértelmezett mérete a szabvány szerint 300×150px, de ezek a számok könnyedén átállíthatók JavaScript segítségével. Mivel a vászon szélességre és magasságára gyakran van szükség rajzolási műveletek során, ezért érdemes ezeket az értékékeket valamilyen konstansban is eltárolni.

```
const szelesseg = 640;
const magassag = 480;
vaszon.width = szelesseg; // szélesség beállítása
vaszon.height = magassag; // magasság beállítása
```

A méretek ismeretében már elkezdhetjük a rajzolást. A legalapvetőbb rajzolóműveletekkel *szakaszokat* és *görbéket* húzhatunk. Később az ezek által kirajzolt útvonalat (*path*) lehetőségünk van megrajzolni (*stroke*) vagy kitölteni (*fill*). A kitöltés és a vonal megrajzolása mindig az éppen aktuális útvonalra vonatkozik. Minden ilyen rajzolási művelethez külön állíthatunk tollszínt, vonalstílust és kitöltési mintázatot. Az útvonalakat a `context` objektum `beginPath` és `closePath` metódusaival tudunk létrehozni.

Példa

```
// háromszög rajzolása
rajz.beginPath(); // elkezdjük az alakzatot
rajz.strokeStyle = "blue"; // a toll színének állítása
rajz.lineWidth = 3; // a toll vastagságának beállítás
rajz.fillStyle = "yellow"; // a kitöltés színének állítása
rajz.moveTo(10, 10); // a 10;10 koordinátájú helyre mozgatjuk a
képzeletbeli tollat
rajz.lineTo(10, 200); // vonal a 10;10-ből a 10;200-ba
rajz.lineTo(200, 200); // vonal a 10;200-ból a 200;200-ba
rajz.lineTo(10, 10); // vonal a 200;200-ból a 10;10-be
rajz.stroke(); // az alakzat körvonalának megrajzolása
rajz.fill(); // az alakzat kitöltése
rajz.closePath(); // az alakzat lezárása
```

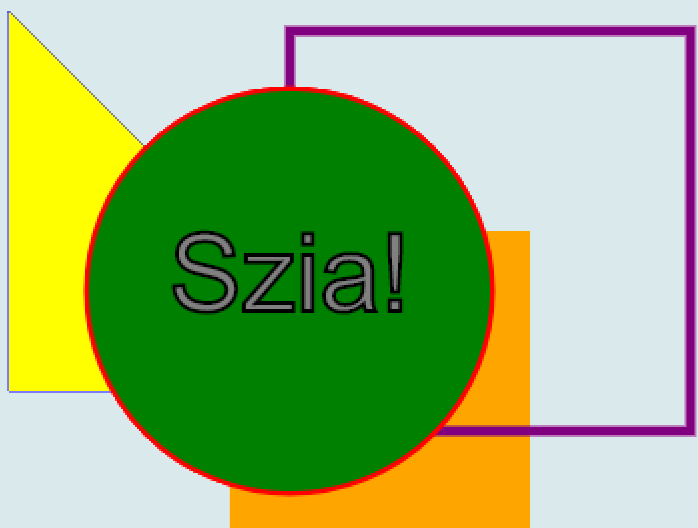


A fenti kód által rajzolt háromszög

A 2D-s rajzoláshoz rendelkezésünkre állnak előre létrehozott rajzoló műveletek is. Ezek segítségével **kör(ív)** ↪, téglalap, szöveg rajzolható a vászonra. Téglalapok és szöveg esetén külön függvények állnak rendelkezésünkre a körvonal és a kitöltés megrajzolására, nem kell az útvonalat kézzel előállítani, ezáltal a `beginPath` és a `closePath` parancsokra sincs szükség ezek rajzolásánál.

Példa

```
// téglalap rajzolása
rajz.strokeStyle = "purple";
rajz.lineWidth = 5;
rajz.fillStyle = "orange";
// 150×150-es téglalap kitöltése a 120;120 bal felső saroktól
rajz.fillRect(120, 120, 150, 150);
// 200×200-as téglalap körvonal rajzolása a 150;150 felső saroktól
rajz.strokeRect(150, 20, 200, 200);

// körív rajzolása
rajz.beginPath();
rajz.strokeStyle = "red";
rajz.lineWidth = 5;
rajz.fillStyle = "green";
// 150;150 középpontú, 100 sugarú körív a 0-tól a 2 Pi szögig, a
szögek radiánban vannak
rajz.arc(150, 150, 100, 0, Math.PI * 2);
rajz.stroke(); // körvonal megrajzolása
rajz.fill(); // kitöltés
rajz.closePath();

// szöveg kiírása
rajz.font = "54px Arial";
rajz.strokeStyle = "black";
rajz.lineWidth = 2;
rajz.fillStyle = "gray";
rajz.fillText("Szia!", 90, 160);
rajz.strokeText("Szia!", 90, 160);
```



A kód által megrajzolt alakzatok

Apróbetűs

Mivel egy-egy alakzat nagyon sok művelettel rajzolható meg, ezért érdemes lehet saját függvényeket bevezetni a gyakori alakzatok megrajzolására.

Példa

```
function haromszog(x1, y1, x2, y2, x3, y3, szin) {
  rajz.beginPath();
  rajz.fillStyle = szin;
  rajz.moveTo(x1, y1);
  rajz.lineTo(x2, y2);
  rajz.lineTo(x3, y3);
  rajz.lineTo(x1, x2);
  rajz.fill();
  rajz.closePath();
}
haromszog(10, 10, 10, 200, 200, 95, "blue");
```

7.1.2 Képek rajzolása vászonra

A vászonra készülő grafikáknál a bonyolultabb alakzatokat sok esetben nem érdemes programozottan előállítani, mivel az rengeteg időt és kódot igényelne. Amennyiben az alakzat fix, nem változik a programunk során, akkor érdemes ezeket az alakzatokat képként megrajzolni valamilyen képszerkesztő programmal, majd a kész képet beszúrni a grafikánkba. Tetszőleges elterjedt formátumú (pl. JPG, PNG, GIF) képfájl felhelyezhető a vászonra.

```
const kep = new Image(); // új kép létrehozása
kep.src = "kepfajl.png"; // kép elérési útjának beállítás
// kép felhelyezése a vászonra az 50;50 koordinátától 100×100px
méretben
rajz.drawImage(kep, 50, 50, 10, 10);
```

Apróbetűs

Kép kirajzolásánál lehetőségünk van arra, hogy ne a teljes eredeti képet rajzoljuk ki, hanem annak csak egy részét. Ebben az esetben a `drawImage` metódust nem 5, hanem 9 paraméterrel kell meghívni, a **dokumentációban** ↪ leírt módon. Ez a változat hasznos lehet az úgynevezett *spritesheet* ↪ alapú animációk megvalósításában.

7.2 Feladatok

1. Készíts segédfüggvényt, ami egy adott (x;y) pontba egy adott sugarú kört rajzol adott színnel!
2. Készítsd el a képen látható ábrát!

Mosolygós arc

```
<canvas></canvas>
<script>
  function $(szelektor) {
    return document.querySelector(szelektor);
  }
  const vaszon = $("canvas");
  const rajz = vaszon.getContext("2d");

  rajz.fillStyle = "gold";
  rajz.beginPath();
  rajz.arc(100, 100, 50, 0, Math.PI * 2);
  rajz.fill();
  rajz.closePath();

  rajz.fillStyle = "black";
  rajz.beginPath();
  rajz.arc(80, 80, 5, 0, Math.PI * 2);
  rajz.arc(120, 80, 5, 0, Math.PI * 2);
  rajz.fill();
  rajz.closePath();

  rajz.lineWidth = 5;
  rajz.strokeStyle = "black";
  rajz.beginPath();
  rajz.arc(100, 100, 30, Math.PI / 4, Math.PI / 4 * 3);
  rajz.stroke();
  rajz.closePath();
</script>
```

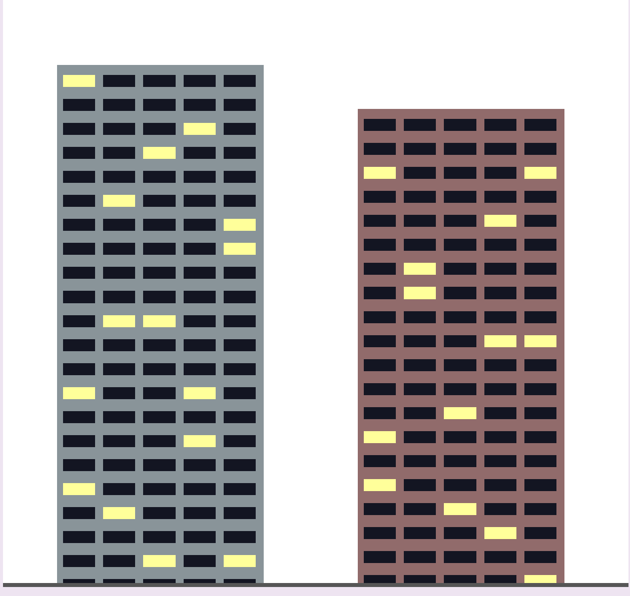
3. Készíts programot, ami megadott százalékos arányban felosztott kördiagramot készít! (körív és vonalak segítségével, a kép illusztráció)

Tortadiagram

```
<canvas></canvas>
<script>
  function $(szelektor) {
    return document.querySelector(szelektor);
  }
  const vaszon = $("canvas");
  const rajz = vaszon.getContext("2d");

  function kordiagram(szazalek) {
    rajz.fillStyle = "red";
    rajz.beginPath();
    rajz.arc(100, 100, 50, 0, Math.PI * 2)
    rajz.fill();
    rajz.closePath();

    rajz.fillStyle = "blue";
    rajz.beginPath();
    const arany = (szazalek / 100) * (Math.PI * 2)
    rajz.moveTo(100, 100);
    rajz.arc(100, 100, 50, 0, arany);
    rajz.fill();
    rajz.closePath();
  }

  kordiagram(15);
</script>
```

4. **Nagyfeladat**

   Készíts programot, ami adott magassággal, adott szélességgel és adott méretű ablakokkal toronyházat rajzol a minta alapján! Próbáld meg megcsinálni, hogy ne minden ablak világítson, csak néhány!

   Toronyházak

Letöltés

⭳ A fejezethez tartozó nagyfeladatok megoldásai ↪

7.3 Események a vásznon

Az eddigiekben megnéztük, hogy hogyan lehet egyszerű ábrákat készíteni a `canvas` elem és a Canvas API segítségével, de ez önmagában még nem egy interaktív program, csupán egy programozott ábrakészítés. Mint az összes többi DOM objektum, a `canvas` is képes eseményeket kiváltani, és ezen események segítségével a felhasználó interakcióba léphet a vászonnal. Leggyakrabban valamilyen egér-eseményt tudunk figyelni a vásznon, vagy pedig az egész oldalra vonatkozó billentyűzet-eseményre tud a vásznon lévő ábra reagálni.

Az egér-események eseményobjektuma számos hasznos információt ad az egér pozíciójáról (`e.offsetX`, `e.offsetY`) és a kattintás esemény milyenségéről (melyik gombbal kattintottunk). Ezek segítségével egyszerűen létrehozhatunk egy primitív rajzolóprogramot, mely a kattintott helyre elhelyez egy pontot a vásznon.

```
<canvas></canvas>
<script>
  /* ... */
  function pontRajzol(e) {
    rajz.beginPath();
    rajz.fillStyle = "black";
    rajz.arc(e.offsetX, e.offsetY, 5, 0, Math.PI * 2); //  az egér
pozíciójába egy 5px sugarú kört rajzolunk
    rajz.fill();
    rajz.closePath();
  }

  vaszon.addEventListener("click", pontRajzol);
</script>
```

Ezzel a módszerrel csak egy-egy pontot tudunk rajzolni. Ha azt szeretnénk, hogy az egérgombot lenyomva folyamatos vonalat tudjunk húzni, akkor az egér állapotát (le van nyomva, nincs lenyomva) el kell tárolni és a rajzolást ettől függően az egér mozgatás eseményhez (`mousemove`) kell kötni. Az állapotot a gomblenyomás (`mousedown`) és gomb felengedés (`mouseup`) események figyelésével tudjuk nyilvántartani.

```
let egerLent = false;
function egerLe() {
  egerLent = true;
}
function egerFel() {
  egerLent = false;
}
function pontRajzol(e) {
  if (egerLent) {
    rajz.beginPath();
    rajz.fillStyle = "black";
    rajz.arc(e.offsetX, e.offsetY, 10, 0, Math.PI * 2);
    rajz.fill();
    rajz.closePath();
  }
}
vaszon.addEventListener("mousemove", pontRajzol);
vaszon.addEventListener("mousedown", egerLe);
vaszon.addEventListener("mouseup", egerFel);
```

Ez a változat már képes folytonos vonalat húzni, ha kellően lassan mozgatjuk az egeret. Mivel az egérmozgatás esemény is csak megadott időközönként érzékeli, hogy az egér elmozdult, így nem garantált, hogy a vonal folytonos lesz. Erre a problémára egy egyszerű megoldás, ha nem csak pontokat rajzolunk, hanem egy vonalat is, az egér előző ismert pozíciójába. Az egér előző ismert

pozíciója könnyen számolható az aktuális pozícióból és az elmozdulásból (`e.movementX`, `e.movementY`).

```
function pontRajzol(e) {
  if (egerLent) {
    rajz.beginPath();
    rajz.fillStyle = "black";
    rajz.arc(e.offsetX, e.offsetY, 5, 0, Math.PI * 2);
    rajz.fill();
    rajz.closePath();
    // vonal az előző ismert pozícióba
    rajz.beginPath();
    rajz.lineWidth = 10;
    rajz.strokeStyle = "black";
    rajz.moveTo(e.offsetX, e.offsetY);
    rajz.lineTo(e.offsetX - e.movementX, e.offsetY - e.movementY);
    rajz.stroke();
    rajz.closePath();
  }
}
```

Egy ilyen rajzolóprogram esetében felmerül az igény, hogy letöröljük a teljes vásznat és új rajzot kezdhessünk. A vászon (egy részének) törlésére rendelkezésre áll a `clearRect` művelet, ami egy téglalap alakú területet töröl. Ha az egész vásznat törölni akarjuk, akkor a (0;0) koordinátáktól egy `szelesseg` × `magassag` méretű téglalapot kell törölni.

```
rajz.clearRect(0, 0, vaszon.width, vaszon.height);
// vagy, ha definiáltuk a megfelelő konstansokat
rajz.clearRect(0, 0, szelesseg, magassag);
```

Ezt a törlő műveletet ha belerakjuk egy függvénybe, akkor könnyen hozzárendelhetjük egy gomb kattintás eseményéhez.

```
<canvas></canvas>
<button>Töröl</button>
<script>
  /* ... */
  function torol() {
    rajz.clearRect(0, 0, szelesseg, magassag);
  }
  $("button").addEventListener("click", torol);
</script>
```

7.4 Feladatok

1. Készíts vezérlőket, amikkel lehet szabályozni a rajzolóprogramban a toll vastagságát és színét! (használhatod az `<input type="number">`, `<input type="color">` vezérlőket)

```
<input type="number" min="1" max="20" value="10">
<input type="color" value="#FF0000">
<br>
<canvas></canvas>
<script>
  function $(szelektor) {
    return document.querySelector(szelektor);
  }
  const vaszon = $("canvas");
  const szelesseg = 640;
  const magassag = 480;
  vaszon.width = szelesseg;
  vaszon.height = magassag;
  const rajz = vaszon.getContext("2d");

  let egerLent = false;
  function egerLe() {
    egerLent = true;
  }
  function egerFel() {
    egerLent = false;
  }

  function pontRajzol(e) {
    if (egerLent) {
      const vastagsag = $("input[type=number]").value;
      const szin = $("input[type=color]").value;
      rajz.beginPath();
      rajz.fillStyle = szin;
       rajz.arc(e.offsetX, e.offsetY, vastagsag / 2, 0, Math.PI *
2);
      rajz.fill();
      rajz.closePath();
      // vonal az előző ismert pozícióba
      rajz.beginPath();
      rajz.lineWidth = vastagsag;
      rajz.strokeStyle = szin;
      rajz.moveTo(e.offsetX, e.offsetY);
             rajz.lineTo(e.offsetX  -  e.movementX,  e.offsetY  -
e.movementY);
      rajz.stroke();
      rajz.closePath();
    }
```

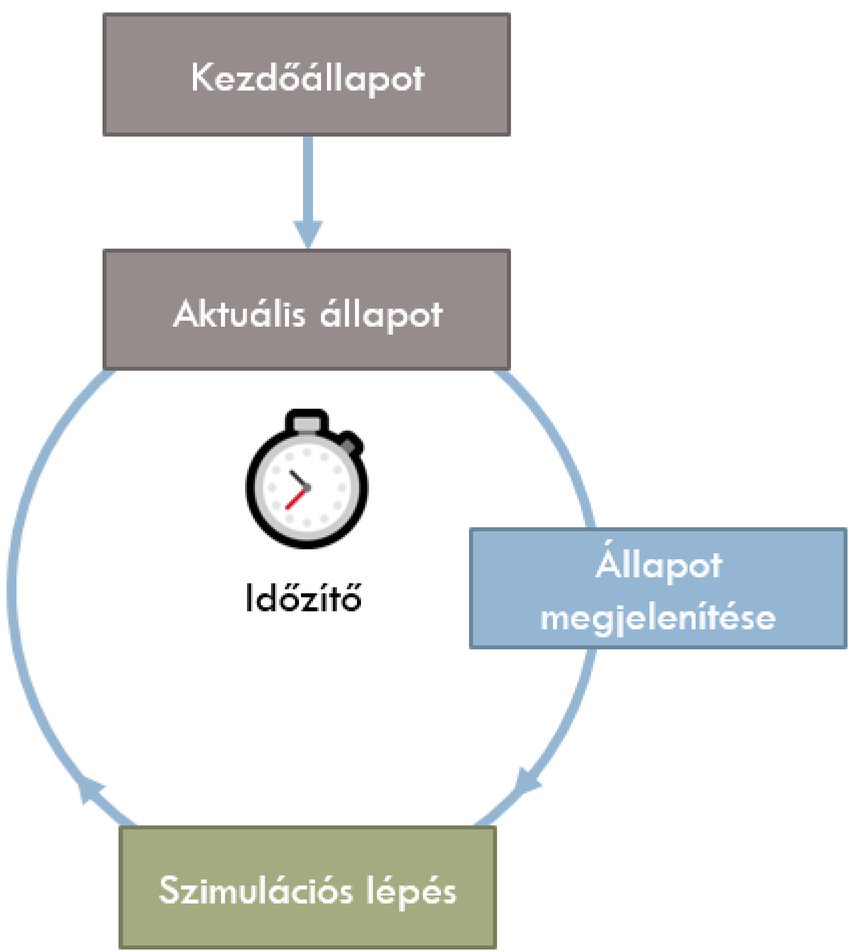
```
  }

  vaszon.addEventListener("mousemove", pontRajzol);
  vaszon.addEventListener("mousedown", egerLe);
  vaszon.addEventListener("mouseup", egerFel);
</script>
```

8 SZIMULÁCIÓK ÉS JÁTÉKOK KÉSZÍTÉSE VÁSZONNAL

8.1 Szimulációk programozása

A vászon programozásával lehetőségünk nyílik nem csak ábrák, rajzok, hanem szimulációk készítésére is. A számítógépes szimulációk lényege az, hogy egy adott állapotot folyamatosan (valamilyen esemény hatására, vagy időzítve) "léptetünk", vagyis egy függvény segítségével kiszámítjuk a jelenlegi állapotból a soron következő állapotot. Időzítés esetén ez a függvény paraméterül kaphatja a két állapot közötti eltelt időt is, így az időfaktort is figyelembe vehetjük a számításban. Az új állapotot előállító függvényt *léptetőfüggvénynek* vagy *szimulációs lépésnek* hívjuk.

A szimulációs lépések egymásutánja létrehoz egy *szimulációs ciklust, melynek folyamatát az alábbi ábra szemlélteti:



A szimulációs ciklus sematikus ábrája

Ahhoz, hogy időzített szimulációkat készítsünk, szükség van a JavaScript nyelv beépített időzítő eljárásainak használatára. A két legegyszerűbb időzítő a `setTimeout` ↪ és a `setInterval` ↪ függvény. A `setTimeout` segítségével adott idővel késleltetve tudunk végrehajtani egy függvényt, míg a `setInterval` adott időközönként folyamatosan ismétel egy függvényt

mindaddig, amíg az időzítőt le nem állítjuk. A `setInterval` függvény alkalmas a szimulációs ciklus működtetésére, a szimulációs lépés adott időközönként történő ismétlésére.

Példa

```
const frissitesiIdo = /*...*/;
let allapot;

function kezdoAllapot() { /*...*/ }
function kovetkezoAllapot() { /*...*/ }

kezdoAllapot();
setInterval(kovetkezoAllapot, frissitesiIdo);
```

Apróbetűs

A modernebb böngészőkben már lehetőség van a `setInterval`-nál precízebb időzítők létrehozására. Erre azért van szükség, mert a `setInterval` nem biztosít pontos időzítést, illetve az így létrejövő animációk tartalmazhatnak villódzást, kimaradhatnak képkockák. Kifejezetten animációkhoz hozták létre a **`requestAnimationFrame`** ↪ függvényt, mely a JavaScript beépített **Date** ↪ típusával kombinálva pontosabb és simább animációt biztosít.

Példa

```
const frissitesiIdo = /*...*/;
let utolsoFrissites = Date.now();
let allapot = kezdoAllapot();

function kezdoAllapot() { /*...*/ }
function kovetkezoAllapot() { /*...*/ }

function animaciosLepes() {
  let most = Date.now();
  if (most > utolsoFrissites + frissitesiIdo) {
    kovetkezoAllapot();
    utolsoFrissites = most;
  }
  requestAnimationFrame(animaciosLepes); // a függvény
önmagára hivatkozik
}
```

Példa

Időzítők és vászon segítségével készíts digitális órát! Használd a JavaScript nyelv beépített `Date` osztályát!

```
<canvas></canvas>
<script>
  const $ = document.querySelector.bind(document);
  const vaszon = $("canvas");
  const rajz = vaszon.getContext("2d")
  function rajzolIdo() {
    const most = new Date();
    const szoveg = most.getHours() + ":" + most.getMinutes() + ":"
+ most.getSeconds();
    rajz.clearRect(0, 0, vaszon.width, vaszon.height);
    rajz.fillStyle = "black";
    rajz.fillText(szoveg, 10, 10);
  }
  setInterval(rajzolIdo, 1000)
</script>
```

Időzítők és a szimulációs ciklus segítségével könnyedén készíthetünk egyszerű fizikai modelleket. Az alapelv nem más, mint a szimulációs lépésben a fizikai törvényszerűségeknek megfelelően beprogramozni az állapotváltozást. Ehhez le kell írni a fizikai modell egy éppen aktuális állapotát melyből számítható a következő állapot az eltelt idő függvényében.

A legegyszerűbb fizikai szimulációk a kinematikai modellek. Ezekben valamilyen test/testek mozgását írjuk le, legegyszerűbb esetben egyszerű mozgásegyenlettel/mozgásegyenletekkel. A szimuláció az alábbi részekre bontható fel:

1. Az éppen aktuális állapotot (hely, sebesség, gyorsulás, **idő**) leíró változók (*aktuális állapot*)
2. Az éppen aktuális állapotot kirajzoló függvény (ami egyben törli az előző állapotot képről)
3. Az állapot változását az eltelt idő ismeretében kiszámító függvény (*szimulációs lépés*)
4. A kezdőállapotot beállító függvény
5. A kezdőállapot beállítása, majd a szimulációs lépés és a kirajzolás időzített ismétlése.

Ezen komponensek segítségével már könnyedén összeépíthető a kész program. A megvalósítás könnyítésére érdemes a kirajzoló függvényt a kezdőállapotot visszaállító függvénybe és a szimulációs lépésbe is közvetlenül beletenni, mert így egyszerűen biztosítható az, hogy ha változik a szimuláció állapota, akkor a változást azonnal látni is fogjuk.

Példa

Nézzük, hogy hogyan szimulálható egy egyszerű ferde hajítás!

1. Először definiáljuk a test helyzetét és mozgását leíró változókat és konstansokat:

```
const g = 9.82; // a gravitációs gyorsulás
const frissitesiIdo = 1000 / 60; // másodpercenként 60 frissítés
let x,  // a test x koordinátája a vásznon
    y,  // a test y koordinátája a vásznon
    vx, // a test x irányú sebessége
    vy; // a text y irányú sebessége
```

Apróbetűs

Ha azt akarjuk, hogy a szimulációnk pontos modellt adjon, akkor ügyelnünk kell a mértékegységekre. Ebben az esetben az 1px = 1m arányítást használva SI mértékegységekben számolunk. Sokszor ahhoz, hogy a szimuláció jól látható legyen valós időben valamilyen módosító távolság- vagy idő aránytényezőt kell használni.

2. Ebben az esetben a kirajzolás nagyon egyszerű, csupán egy kört rajzolunk a megfelelő pozícióba.

```
function kirajzol() {
  rajz.clearRect(0, 0, szelesseg, magassag); // a vászon törlése
  rajz.beginPath();
  rajz.fillStyle = "black";
  rajz.arc(x, y, 5, 0, Math.PI * 2); // 5 px sugarú fekete kör (x;y)-ba
  rajz.fill();
  rajz.closePath();
}
```

3. A fizikai paraméterek alapján már leírható a test mozgása függvény formájában:

```
function kovetkezoAllapot() {
  // eltelt idő a legutóbbi állapot óta másodpercben
  const dt = frissitesiIdo / 1000;
  x += vx * dt; // vízszintesen egyenletes mozgás
  y += vy * dt + (g / 2 * dt * dt); // függőlegesen egyenletesen gyorsuló mozgás
  vy += g * dt; // függőleges irányú egyenletes gyorsulás
  kirajzol();
}
```

4. A kezdőállapotot beállító függvény:

```
function kezdoAllapot() {
   x = szelesseg / 10; // vízszintesen a vászon tizedétől
indulunk
   y = magassag / 2;    // függőlegesen a vászon felétől
indulunk
  vx = 20;            // 20px/s vízszintes kezdősebesség
   vy = -20;             // 20px/s függőleges kezdősebesség
felfelé
  kirajzol();
}
```

5. A szimuláció indítása:

```
kezdoAllapot();
setInterval(kovetkezoAllapot, frissitesiIdo);
```

Apróbetűs

Fizikai könyvtárak

Számos olyan JavaScript függvénykönyvtár létezik, melyek a canvas grafikában a fizika megvalósításáért felelősek. Ilyen például a **Matter.js ↪** vagy a **PhysicsJS ↪**.

8.2 Feladatok

1. Módosítsd a ferde hajítás programját, hogy a labda visszapattanjon a vászon szélein!

```
/* ... */
function kovetkezoAllapot() {
  const dt = frissitesiIdo / 1000;
  x += vx * dt;
  y += vy * dt + (g / 2 * dt * dt);
  vy += g * dt;
  // visszapattanás
  if (x <= 0 || x >= szelesseg) {
    vx *= -1;
  }
  if (y <= 0 || y >= magassag) {
    vy *= -1;
  }

  kirajzol();
}
/* ... */
```

2. **Nagyfeladat**

Készíts szimulációt, melybe minden 50. szimulációs lépésben felülről véletlenszerű helyen elkezd esni lassan egy hópehely, és amikor eléri a vászon alját, akkor ott megáll!

8.3 Játékok programozása

A szimulációk programozásához nagyon közelálló témakör az egyszerűbb játékok programozása. Egy játék működésének alapelve nagyban hasonlít a korábban bemutatott szimulációs ciklushoz, csupán pár különbség van. Az játékok készítésével kapcsolatos koncepciókat egy egyszerű aszteroida-kikerülős játék példáján mutatjuk be. A kiindulási alapunk a szimulációs ciklus, az ehhez képesti különbségeket mutatjuk be.

Példa

Milyen változók írják le a játékunk éppen aktuális állapotát?

```
const urhajoY = magassag / 8 * 7; // az űrhajó fix y koordinátája
const urhajoSugar = 10; // az űrhajót megtestesítő kör sugara
const aszteroidaSebesseg = 4; // az aszteriodák sebessége
(px/lépés)

let aszteroidak; // aszerodiákat tároló tömb
// egy aszteroidát egy {x, y} koordináta ír le és a sugara (r) ír
le
```

```
let urhajoX; // az űrhajó x koordinátája
```

A játék állapotának kirajzolása két részből áll, az űrható és az aszteroidák kirajzolása.

```
function kirajzol() {
  rajz.clearRect(0, 0, szelesseg, magassag); // a vászon törlése
  rajz.fillStyle = "gray";
  // végigmegyünk az aszterodiákon
  for (let aszteroida of aszteroidak) {
    // kirajzolunk egy aszeroidát
    rajz.beginPath();
    // `aszteroida.sugar` sugarú kör (x;y)-ba
    rajz.arc(aszteroida.x, aszteroida.y, aszteroida.sugar, 0,
Math.PI * 2);
    rajz.fill();
    rajz.closePath();
  }
  // az űrhajó kirajzolása
  rajz.fillStyle = "red";
  rajz.beginPath();
  // `urhajoSugar` sugarú kör (x;y)-ba
  rajz.arc(urhajoX, urhajoY, urhajoSugar, 0, Math.PI * 2);
  rajz.fill();
  rajz.closePath();
}
```

A szimulációs lépésben csupán az aszteroidák mozognak a fix sebességükkel.

```
function kovetkezoAllapot() {
  for (let aszterodia of aszteroidak) {
    aszterodia.y += aszteroidaSebesseg;
  }
  kirajzol();
}
```

A kezdőállapot nem más, min az, hogy nincsnek aszteroidák, és az űrhajó középen van.

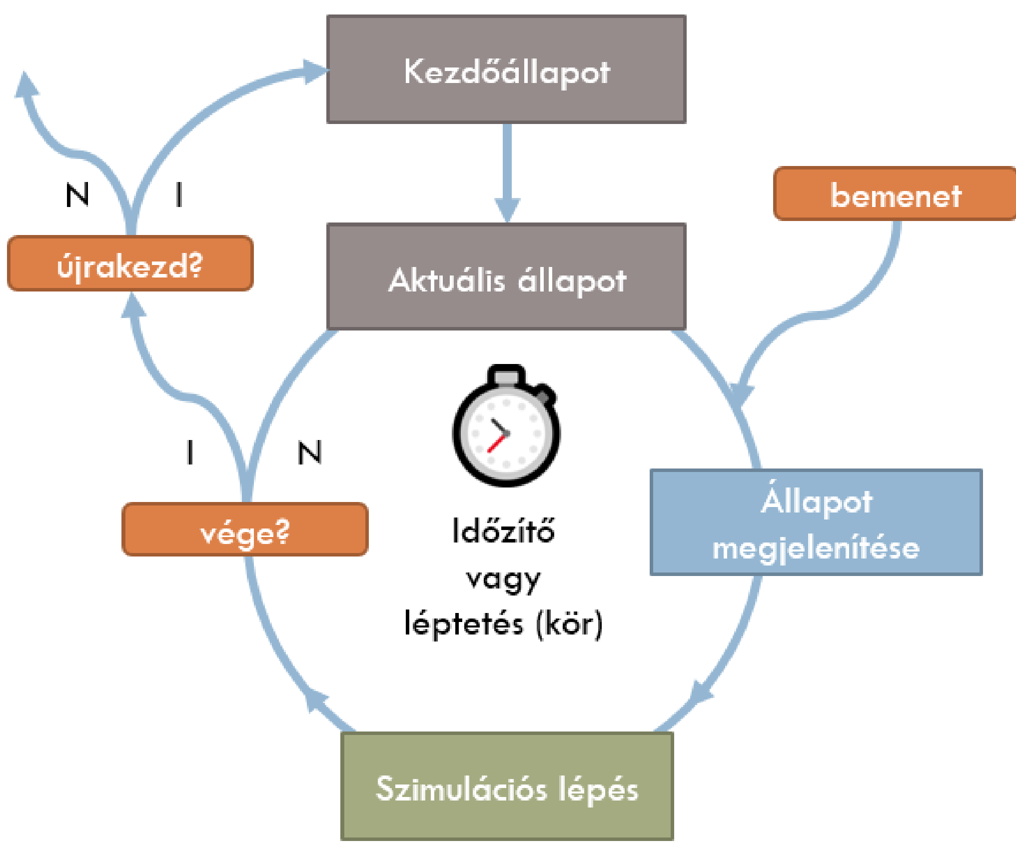
```
function kezdoAllapot() {
  aszteroidak = [];
  urhajoX = szelesseg / 2;
  kirajzol();
}
```

A játékciklust a szimulációhoz hasonlóan elindítjuk:

```
kezdoAllapot();
setInterval(kovetkezoAllapot, frissitesiIdo);
```

Apróbetűs

Ahhoz, hogy ne kelljen egy idő után túl sok aszteroidával számolnia a programnak érdemes törölni az adatszerkezetből azokat az aszteroidákat, amik már teljesen elhagyták a vásznat. Ennek hiányában a játék egy idő után elkezd lassulni a túl nagy számítási igény miatt.



A játékciklus sematikus ábrája

Ahogy az a játékciklust bemutató ábrán látható, egy játék alapvetően két fő ponton különbözik az egyszerű szimulációktól:

- A felhasználó (játékos) valamilyen módon interakcióba lép a játékkal, annak lefolyására hatással van valamilyen bemenet révén
- A játék valamilyen formában (győzelem/vereség) előbb-utóbb véget ér (sok olyan szimuláció is létezik, aminek van végállapota).

Ezen különbségek alapján mondhatjuk, hogy a játék nem más, mint egy véges, interaktív szimuláció. Az időzítő fajtájától függően (léptetés vagy valós idejű időzítés) megkülönböztetünk *valós idejű* és *körökre osztott* játékokat.

Valós idejű játékok (mint amilyen példánkban is szerepel) esetén a játék akkor is előrehalad az időzítőnek megfelelően, ha a felhasználótól semmilyen bemenet nem érkezik. Ebben az esetben az idő alapú *időzítő mellett*, a játékos bizonyos *eseményekkel tud beleavatkozni* a játékba, ami a következő cikluslépésben fejti ki a hatását.

Körökre osztott játékokban maga a cikluslépés is valamilyen felhasználói interakció következményeképp jön létre. Jó példa erre az aknakereső játék, ahol a felhasználó kattintásának hatására jön létre az új játékállapot, vagyis az új “kör”.

Ahhoz, hogy a játékos interakcióba lépjen a játékkal, ahhoz valamilyen esemény segítségével kommunikálnia kell a játékkal. Ez lehet bármilyen jellegű esemény akár billentyűlenyomás, akár az egér mozgatása, kattintás. Ezekhez az eseményekhez tartozó eseménykezelőket a játékciklustól függetlenül tudjuk regisztrálni, ezek csupán a játék aktuális állapotát módosítják, a kirajzolásról a játékciklus következő lépésében gondoskodunk.

Példa

Az űrhajó mozgatása billentyű lenyomására.

```
function gombLe(e) {
  e.preventDefault();
  if (e.key == "ArrowLeft") { // ha a bal gombot nyomtuk
    urhajoX -= 5;
  } else if (e.key == "ArrowRight") { // ha a jobb gombot nyomtuk
    urhajoX += 5;
  }
}

// hozzárendeljük a `gombLe` függvényt a billenytűlenyomáshoz
window.addEventListener("keydown", gombLe);
```

Apróbetűs

Megfigyelhetjük, hogy ezzel a megvalósítással az űrhajó igencsak szaggatottan mozog. Ennek az az oka, hogy a gomb lenyomás eseményt az operációs rendszer nem folyamatosnak, hanem szakaszosnak érzékeli. Ha szeretnénk, hogy egyenletesen mozogjon a hajó, akkor nyilván kell tartani egy változóban, hogy le van-e éppen nyomva az irányító gomb és a szimulációs lépésben kell változtatni az űrhajó koordinátáit a vezérlőgombok állapotának függvényében.

Sok játékban a változatosságot valamilyen időközönként bekövetkező véletlen esemény biztosítja. Az ilyen időzített véletlen eseményeket köthetjük egy a játékciklus üzemeltető időzítőtől független másik időzítőhöz. Tulajdonképpen ebben az esetben sincs másról szó, mint egy eseményre történő reagálásról, csupán a kiváltó esemény egy időzítő lejárása.

Példa

Bizonyos időközönként létrejönnek véletlenszerű sugarú aszteroidák véletlenszerű helyen.

```
const ujAszteroidaIntervallum = 1000; // másodpercenként egy új
aszteroida

function ujAszteroida() {
  aszteroidak.push({
    x: veletlenEgesz(0, szelesseg),
    y: -20, // a vásznon kívül jön létre, hogy ne megjelenjen,
hanem beússzon,
    sugar: veletlenEgesz(10, 30)
  });
}

setInterval(ujAszteroida, ujAszteroidaIntervallum);
```

A játék során a játék aktuális állapota alapján mindig egyértelműen meghatározható, hogy a játék véget ért-e győzelemmel vagy vereséggel. Ezt a vizsgálatot minden cikluslépésben el kell végezni, és amennyiben bekövetkezik valamelyik kimenet feltétele, úgy a játékban ezt jelezni kell - jellemzően a játék megállításával és valamilyen üzenettel, pontszámmal. Ha véget ért a játék, akkor lehetőség van az újraindításra is, erre rendelkezésre áll a szimulációs példákban is bemutatott `kezdoAllapot` függvény.

Példa

Akkor van vége a játéknak, ha valamelyik aszteroidával ütköztünk.

```
function kovetkezoAllapot() {
  /* ... */
  // minden lépésben ellenőrizni kell, hogy vége van-e a játéknak
  vegeEllenoriz();
  /* ... */
}

function utkozik(aszteroida) {
  // akkor ütközik, ha a két középpont távolsága kisebb, mint a
sugarak összege
  const tavolsag = Math.sqrt(
                    Math.pow(urhajoX - aszteroida.x, 2) +
                    Math.pow(urhajoY - aszteroida.y, 2)
                  );
  return tavolsag < (urhajoSugar + aszteroida.sugar);
}
```

```
function vegeEllenoriz() {
  // akkor van vége, ha létezik olyan aszteroida, amivel ütközünk
  let vege = aszteroidak.some(aszteroida => utkozik(aszteroida));
  if (vege) {
    // a játék vége itt most annyit jelent, hogy kapunk egy
felugró ablakot és utána azonnal újraindul
    alert("Vége a játéknak");
    kezdoAllapot();
  }
}
```

Apróbetűs

A példában bemutatott vesztés esetén azonnal újraindítás nem túl felhasználóbarát. Érdemesebb ilyenkor megállítani a játékot. Ehhez az időzítők indításakor el kell tárolni a `setInterval` függvény visszatérési értékét és ezzel az értékkel meghívni a `clearInterval` függvényt, ami leállítja az időzítőt. Ezt az összes játékban lévő időzítőre meg kell tenni, mert ha úgy indítunk újra egy időzítőt, hogy előtte nem állítottuk le, akkor kétszer fog futni párhuzamosan. A játék újraindítását egy tetszőleges eseményre (pl. egy adott billentyű lenyomása) végezhetjük el.

```
let idozito;

function vegeEllenoriz() {
  /* ... */
  if (/* végfeltétel */) {
    clearInterval(idozito);
    /* ... */
  }
}

function start() {
  kezdoAllapot();
  idozito = setInterval(/* ... */);
}

// valamilyen esemény (pl. egy gomb megnyomása) hatására
indul el/újra a játék
/* ... */.addEventListener(/* valamilyen esemény */, start);
```

Apróbetűs

Külső könyvtárak játékfejlesztéshez

Számos külső könyvtár, játékmotor létezik JavaScript nyelven, amivel magasabb szinten, sokkal gyorsabban lehet komplexebb játékokat készíteni. Ilyen JavaScript könyvtárakat gyűjtöttek össze a **GitHub egyik gyűjtényében ↪**.

8.4 Feladatok

1. **Nagyfeladat**

   Készítsd el a Flappy Bird játék saját változatát! Használhatod akadálynak a korábban készített toronyházakat! Használd az aszteroidás példában látottakat a megoldáshoz!

Letöltés

- ⭳ **A fejezethez tartozó nagyfeladatok megoldásai ↪**
- ⭳ **A helikopteres nagyfeladat teljes megoldása ↪**

9 ADATOK MENTÉSE A BÖNGÉSZŐBEN

A korábbi fejezetekben megtanultuk, hogy hogyan készíthetünk kliensoldali webes technológiák segítségével egyszerűbb webes alkalmazásokat, játékokat. Ezek az alkalmazások vagy valamilyen űrlapelemek, vagy vásznon megjelenő grafikák segítségével működtek. Minden esetben az alkalmazás rendelkezett valamilyen állapottal (pl. aknakereső esetében a pálya jelenlegi állása, a `canvas` játéknál a pontszám vagy az akadályok helyzete), ezt az állapotot változókban tároltuk. Ehhez kapcsolódó hiányossága az eddig készített programoknak, hogy mivel az állapot egyszerű változókban van tárolva, nem képesek ennek az állapotnak a hosszabb távú tárolására (idegen szóval perzisztálására).

Miért lenne jó, ha tudnánk adatot tárolni? Az állapot perzisztálásával számos plusz funkcióval lehet kiegészíteni egy webes alkalmazást, mivel az képessé válik a futások között is megőrizni valamilyen információt (például egy játék esetében az eddig elért legmagasabb pontszámot, vagy a legjobban teljesítő játékosok neve, vagy mondjuk egy bevásárlólistát nyilvántartó alkalmazásban a lista elemeit, vagy bármilyen hosszú távon szükséges beállítást). Az adatok tárolására számos lehetőség van böngészőben futó programok esetében. Ezek a teljesség igénye nélkül az alábbiak:

- Adatok tárolása a böngészőprogram által hosszútávra (`localStorage`)
- Adatok tárolása a böngészőprogram által egy munkamenet erejéig (`sessionStorage`)
- Adatok tárolása a böngészőprogram által indexelt adatbázisban (`indexedDB`)
- Adatok tárolása külső szolgáltatás által, valamilyen távoli adatbázisban

Ezen lehetőségek közül a `localStorage` az, aminek kellőképpen egyszerű, és a legtöbb egyszerű perzisztálási feladat megoldására alkalmas lehet. Előnye, hogy egyszerű interfésze révén könnyen használható, hátránya viszont, hogy mivel a helyi gépen tárol információt, ezért más számítógépről ezek az adatok nem elérhetőek.

> **Apróbetűs**
>
> Ha azt szeretnénk, hogy az elmentett adatok bármilyen számítógépről elérhetőek legyenek, akkor valamilyen külső szolgáltatást, távoli adatbázist kell használnunk. Egy könnyen és kényelmesen használható ilyen szolgáltatás a **Google Firebase** ↪ valós idejű adatbázisszolgáltatása, de akár egy egyszerű **Google Táblázatot** ↪ is lehet használni adatbázisként.
>
> A legtöbb ilyen szolgáltatás valamilyen programozási interfészen (API) érhető el megfelelő biztonsági kulcsokat használva. Bizonyos szolgáltatások előre előkészített függvénykönyvtárakat is biztosítanak, melyek megkönnyítik a külső adatbázis használatát.

9.1 A `localStorage` használata

A `localStorage` **API** ↪ könnyű hozzáférést biztosít a böngészőben történő adattárolásra. Az eltárolt adatokat úgy a legegyszerűbb elképzelni, mint egy egyszerű JavaScript objektumot, mely név-érték párokat tartalmaz. A `localStorage` objektumot a globálisan elérhető `window` objektumon keresztül tudjuk elérni.

```
const tarolo = window.localStorage;
```

A `localStorage` objektumot tudjuk használni úgy, mint egyszerű JavaScript objektumot, de a javasolt módszer a **Web Storage API** ↪ használata. Értékek olvasása, írása és törlése a `getItem`, `setItem` és `removeItem` metódusokkal történik.

```
tarolo.setItem("nev", "ertek");
tarolo.getItem("nev"); // "ertek"
tarolo.removeItem("nev");
tarolo.getItem("nev"); // null
```

Ezzel a módszerrel tetszőleges egyszerű értéket (szám, szöveg, logikai) tárolhatunk a `localStorage` segítségével. Összetett adatszerkezetek mentésére ilyen formában nem alkalmas a `localStorage`, az ilyen mentett adatokat nem lehet visszanyerni.

```
const osszetettAdat = {
  mezo1: "ertek",
  mezo2: 100,
  mezo3: [true, false]
};
tarolo.setItem("osszetettAdat", osszetettAdat);
tarolo.getItem("osszetettAdat"); // "[object Object]"
```

Tömbök, objektumok mentéséhez azokat sorosítani (idegen szóval szerializálni) kell. Ez annyit tesz, hogy valamilyen olyan szöveges formára kell őket hozni, ami már tárolható, és amiből visszanyerhető az eredeti adat. Erre a problémára egyszerű megoldást ad a JSON formátum, amit kifejezetten ilyen formátumú adatok szöveges tárolására találtak ki. A szöveges formátumra alakításhoz a `JSON.stringify`, míg a visszaalakításra a `JSON.parse` függvényt használhatjuk.

```
const osszetettAdat = {
  mezo1: "ertek",
  mezo2: 100,
  mezo3: [true, false]
};
tarolo.setItem("osszetettAdat", JSON.stringify(osszetettAdat));
JSON.parse(tarolo.getItem("osszetettAdat")); // { mezo1: "ertek",
mezo2: 100, mezo3: [ true, false ] }
```

9.2 Feladatok

1. **Nagyfeladat**

   Egészítsük ki a korábban készített játékainkat, hogy azok alkalmasak legyenek az eddigi legmagasabb pontszám tárolására és megjelenítésére a `localStorage` segítségével.